

Cours MALG & MOVEX

Overview of the course

Dominique Méry
Telecom Nancy, Université de Lorraine
(30 mars 2025 at 9:48 P.M.)

Année universitaire 2024-2025

- 1 Tracking bugs in C codes
- 2 Requirement engineering
- 3 Introduction by Example
 - Detecting overflows in computations
 - Computing the velocity of an aircraft on the ground
 - Computing the date of Easter
 - Computing how many days between two dates
- 4 Context and Objectives
- 5 Testing Phase
- 6 The Cleanroom Model
- 7 Verification of program properties
- 8 Topics of course

① Tracking bugs in C codes

② Requirement engineering

③ Introduction by Example

- Detecting overflows in computations

- Computing the velocity of an aircraft on the ground

- Computing the date of Easter

- Computing how many days between two dates

④ Context and Objectives

⑤ Testing Phase

⑥ The Cleanroom Model

⑦ Verification of program properties

⑧ Topics of course

Problem versus Solution



Examples of bugs in C codes

- ▶ bug0.c prints *Result : 1*

Examples of bugs in C codes

- ▶ bug0.c prints *Result : 1*
- ▶ bug00.c prints ... *Result : x= 94; Result : i=100000 15*

- ▶ bug0.c prints *Result : 1*
- ▶ bug00.c prints ... *Result : x= 94; Result : i=100000 15*
- ▶ bug000.c prints *Floating point exception : 8* and is a version modified from bug00.c : `srand(time(NULL)+i);`

- ▶ bug0.c prints *Result : 1*
- ▶ bug00.c prints ... *Result : x= 94; Result : i=100000 15*
- ▶ bug000.c prints *Floating point exception : 8* and is a version modified from bug00.c : `srand(time(NULL)+i);`
- ▶ *bug1.c* functional bug

- ▶ bug0.c prints *Result : 1*
- ▶ bug00.c prints ... *Result : x= 94; Result : i=100000 15*
- ▶ bug000.c prints *Floating point exception : 8* and is a version modified from bug00.c : `srand(time(NULL)+i);`
- ▶ *bug1.c* functional bug
- ▶ *bug2.c* stupid bug of writing a code

- ▶ bug0.c prints *Result : 1*
- ▶ bug00.c prints ... *Result : x= 94; Result : i=100000 15*
- ▶ bug000.c prints *Floating point exception : 8* and is a version modified from bug00.c : `srand(time(NULL)+i);`
- ▶ *bug1.c* functional bug
- ▶ *bug2.c* stupid bug of writing a code
- ▶ bug5.c prints *Element 10 found at index 5*

Listing 1 – Bug bug0

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y;
    // Seed the random number generator with the current time
    srand(time(NULL));
    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: -%d\n", y);
    return 0;
}
```

Listing 2 – Bug bug00

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y, i=0;

    for (i = 0; i <= 100000; i++) {
        // Seed the random number generator with the current time
        srand(time(NULL));

        // Generate a random number between 1 and 100
        x = rand() % 100 + 1;
        printf("Result: -x=-%d\n", x);
        // Perform some calculations
        y = x / (100 - x);

        printf("Result: -i=%d - -%d\n", i, y);
    }

    return 0;
}
```

Listing 3 – Bug bug000

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y, i=0;

    for (i = 0; i <= 100; i++) {
        // Seed the random number generator with the current time
        srand(time(NULL)+i);

        // Generate a random number between 1 and 100
        x = rand() % 100 + 1;
        printf("Result: -x=-%d\n", x);
        // Perform some calculations
        y = x / (100 - x);

        printf("Result: -i=%d - y=%d\n", i, y);
    }

    return 0;
}
```


Listing 5 – Bug bug2

```
#include <stdio.h>

int main() {
    int x = 5;
    int y = 3;

    // Bug 1: Incorrect variable in the printf statement
    printf("The-value-of-x-is: -%d\n", y);

    // Bug 2: Infinite loop
    while (x > 0) {
        printf("x-is-greater-than-0\n");
    }

    return 0;
}
```

Listing 6 – Bug bug5

```
#include <stdio.h>

int binarySearch(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        // Bug 1: Incorrect comparison
        if (arr[mid] = target) { // Should be '==' for comparison, not '='
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1; // Bug 2: Update 'left' incorrectly
        } else {
            // Bug 3: Missing update for 'right' in the else case
        }
    }

    return -1;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 78};
    int n = sizeof(arr) / sizeof(arr[0]);

    int target = 10;

    int result = binarySearch(arr, n, target);

    if (result != -1) {
        printf("Element-%d-found-at-index-%d\n", target, result);
    } else {
        printf("Element-%d-not-found-in-the-array\n", target);
    }
}
```

Listing 7 – Bug bug6

```
#include <stdio.h>
#include <limits.h>

int divide(int a, int b) {
    if (b != 0) {
        return a / b; // Bug: Does not handle overflow
    } else {
        return 0;
    }
}

int main() {
    int x = INT_MIN;
    int y = -1;
    int result = divide(x, y);

    printf("Result: %d\n", result);

    return 0;
}
```

- ▶ Defining the problem : *computing the average of two integer numbers, computing the date of Easter for a given year, controlling the access of people in a given location, managing a vote session, ...*
- ▶ Analysing the problem : Is the problem already known ? What are the inputs and the outputs of the problem ? What are the entities involved in the problem ? What is the domain problem (regulations, standards, architectures, ...) ?
- ▶ Developing possible solutions : prototypes ? simulations ?
- ▶ Evaluating options : scenarios ? questioning the customer ?
- ▶ Selecting the best options : questioning the customer ? playing with prototypes ?
- ▶ Developing the solution
- ▶ Assessing the results.



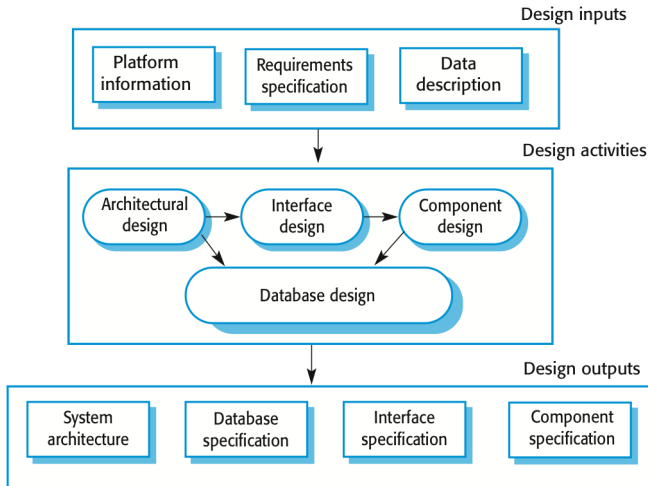
(from Software Engineering by Ian Sommerville)

Requirement Engineering

The process of establishing what services are required and the constraints on the system's operation and development.

- ▶ Feasibility study : Is it technically and financially feasible to build the system ?
- ▶ Requirements elicitation and analysis : What do the system stakeholders require or expect from the system ?
- ▶ Requirements specification : Defining the requirements in detail
- ▶ Requirements validation : Checking the validity of the requirements

General Model of the Design Process



(from Software Engineering by Ian Sommerville)

Listing 8 – Function average

```
#include <stdio.h>
#include <limits.h>
int average(int a,int b)
{
    return ((a+b)/2);
}

int main()
{
    int x,y;
    x=INT_MAX;y=INT_MAX;
    printf(" Average - - for -%d- and -%d- is -%d\n" ,x,y ,
          average(x,y));
    return 0;
}
```


Execution produces a result

Average for 2147483647 and 2147483647 is -1

Execution produces a result

Average for 2147483647 and 2147483647 is -1

Using frama-c produces a required annotation

```
int average(int a, int b)
{
    int __retres;
    /*@ assert rte: signed_overflow: -2147483648 <= a + b; */
    /*@ assert rte: signed_overflow: a + b <= 2147483647; */
    __retres = (a + b) / 2;
    return __retres;
}
```

Listing 9 – Function average.....

```
#include <stdio.h>
#include <limits.h>
/*@ requires 0 <= a;
    requires a <= INT_MAX ;
    requires 0 <= b;
    requires b <= INT_MAX ;
    requires 0 <= a+b;
    requires a+b <= INT_MAX ;
    ensures \result <= INT_MAX;
*/
int average(int a,int b)
{
    return((a+b)/2);
}

int main()
{
    int x,y;
    x=INT_MAX / 2;y=INT_MAX / 2;
    // printf("Average for %d and %d is %d\n",x,y,
    //      );
    return average(x,y);
}
```



- ▶ Estimated ground velocity of the aircraft should be available only if it is within 3 km/hr of the true velocity at some moment within past 3 seconds

Characterization of a System (I)

- ▶ **NG velocity system :**
 - **Hardware :**
 - ▶ *Electro-mechanical sensor* : detects rotations
 - ▶ *Two 16-bit counters* : Rotation counter, Milliseconds counter
 - ▶ *Interrupt service routine* : updates rotation counter and stores current time.
 - **Software :**
 - ▶ *Real-time operating system* : invokes update function every 500 ms
 - ▶ *16-bit global variable* : for recording rotation counter update time
 - ▶ *An update function* : estimates ground velocity of the aircraft.
- ▶ **Input data available to the system :**
 - *time* : in milliseconds
 - *distance* : in inches
 - *rotation angle* : in degrees
- ▶ **Specified system performs velocity estimations in *imperial* unit system**
- ▶ **Note :** expressed functional requirement is in *SI* unit system (km/hr).

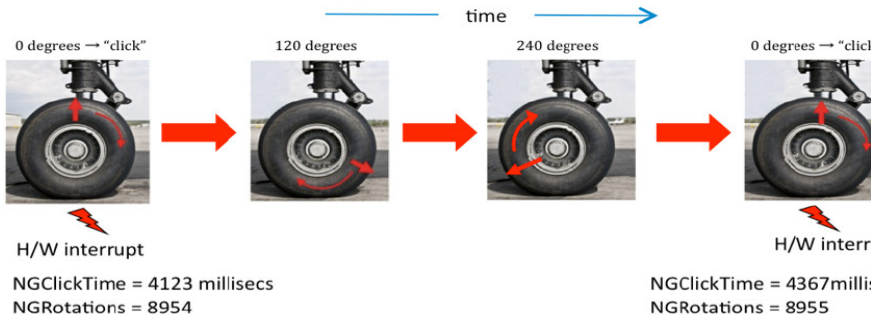
What are the main properties to consider for formalization ?

- ▶ Two different types of data :
 - counters with modulo semantics
 - non-negative values for time, distance, and velocity
- ▶ Two dimensions : *distance* and *time*
- ▶ Many units : distance (inches, kilometers, miles), time (milliseconds, hours), velocity (kph, mph)
- ▶ And interaction among components

How should we model ?

- ▶ Designer needs to consider units and conversions between them to manipulate the model
- ▶ One approach : Model units as *sets*, and conversions as constructed types – *projections*.
- ▶ Example :
 - 1 $estimateVelocity \in \text{MILES} \times \text{HOURS} \rightarrow \text{MPH}$
 - 2 $mphTokph \in \text{MPH} \rightarrow \text{KPH}$

Sample Velocity Estimation



WHEEL_DIAMETER = 22 inches
PI = 3.14

12 inches/foot
5280 feet/mile

estimatedGroundVelocity = distance travel/elapsed time
= $((3.14 * 22) / (12 * 5280)) / ((4367 - 4123) / (1000 * 3600))$
= 16 mph

Safety Property

- ▶ Storing the number of `NGClick` in a n-bit variable `VNGClick`
- ▶ Integers are denoted by the set *Int* and is simply defined by the interval $Int \hat{=} INT_MIN..INT_MAX$.
- ▶ Safety requirement :
*The value of `VNGClick` is always in the range of implementation *Int* or equivalently $VNGClick \in Int$*

$$RTE_VNGClick : 0 \leq vNGClick \leq INT_MAX \quad (1)$$

- ▶ The current value of `VNGClick` is always bounded by the two values 0 and `INT_MAX`.

Computing the date of easter

Easter is the Sunday following the first full moon of spring, that is, according to the definition established by the Council of Nicea in 325 :
"Easter is the Sunday following the 14th day of the Moon which reaches that age on or immediately after March 21."

According to this definition, Easter falls between March 22 and April 25 of each year.

- ▶ Easter date 2019 : 20 march / 21 avril
- ▶ Easter date 2020 : 7 april / 12 april
- ▶ Easter date 2021 : 28 march / 4 april
- ▶ Easter date 2022 : 16 april / 17 april 2022
- ▶ Easter date 2023 : 6 april / 9 april 2023
- ▶ Easter date 2024 : 25 march / 31 march 2024
- ▶ Easter date 2025 : 12 april / 20 april 2025
- ▶ Easter date 2026 : 2 april / 5 april 2026

- ▶ The year is given as a number year
- ▶ The date of Easter is computed as a date day and month

General Protocol for Computing the Date of Easter

- ▶ Easter is a Sunday.
- ▶ Date of the spring full moon `full-moon-day` and `full-moon-month` from the 21st of March.
- ▶ `full-moon-month` is either 3 or 4.
- ▶ The earliest date is March 22nd
- ▶ Determine the First Sunday just after the spring full moon

Listing 10 – Using C function for date of Easter

```
*****  
* year * day * month *  
*****  
* 2019 * 21 * April *  
*****  
* 2020 * 12 * April *  
*****  
* 2021 * 4 * April *  
*****  
* 2022 * 17 * April *  
*****  
* 2023 * 9 * April *  
*****  
* 2024 * 31 * March *  
*****  
* 2025 * 20 * April *  
*****  
* 2026 * 5 * April *  
*****  
*****
```

- ▶ The day of Spring namely March 21 : $dspring = (21, 3)$ (**domain property**)
- ▶ The table of Full Moon of the current year : $tfullmoon[NFM]$ (**domain property**)
- ▶ Computing the date of the first full moon of Spring :
 $dfullmoon = tfullmoon[i]$ where $i \in 0..NFM-1$ (**algorithmic property**)
- ▶ Computing the date of the next Sunday starting from
 $nextday(dfullmoon)$ (**algorithmic property**)

Tryptich \mathcal{D}, \mathcal{A} sat \mathcal{R}

- ▶ \mathcal{D} (domain of problem) contains informations on Moon Phases.
- ▶ \mathcal{R} (requirements or prescription) expresses the day of Easter following the rule.
- ▶ \mathcal{A} (algorithm or program or protocol) contains the different algorithms used for computing necessary informations.

- ▶ Verification : techniques and tools applied to check the *internal* relationship between the three components $\mathcal{D}, \mathcal{A}, \mathcal{R}$
- ▶ Validation : techniques and tools applied to check the *external* relationship between the three components $\mathcal{D}, \mathcal{A}, \mathcal{R}$ and external entities.

How many days for the next event ?

Computing the number of days for waiting a given future date ?

You may want to get the number of days between the current day and a future day :

- ▶ Number of days until Christmas
- ▶ Number of days until Retirement
- ▶ Notification of an event three days before a given date.
- ▶ ...

What is the next day of a given date ?

- ▶ Domain Properties related to dates and calendars :
 - A year is made up of 12 months
 - There are two kinds of months 30 days and 31 days but one month has 28 days and 29 days when the year is a leapyear.
- ▶ Mathematical definition of a leapyear

Listing 11 – pre/post specification for leapyear(int year)

```
#ifndef _LEAP_H
/*@ requires -5000 <= year;
   @ ensures year % 400 == 0 ==> \result == 1;
   @ ensures year % 400 != 0 && year % 100 != 0 && year % 4 == 0 ==> \result == 1;
   @ ensures year % 400 != 0 && year % 100 != 0 && year % 4 != 0 ==> \result == 0;
   @ ensures \result == 1 ==> (year % 400 == 0) || (year % 400 != 0 && year % 100 != 0 && year % 4 == 0);
   @ ensures \result == 0 ==> (year % 400 != 0) && (year % 400 == 0 || year % 100 == 0 || year % 4 != 0);
*/
int leapyear(int year);
#endif
```


Defining leapyear(int year)

Listing 12 – Using C function for date of Easter

```
#include "leapyear.h"

int leapyear(int year)
{
    int r=0; // = 0;
    // leap year if perfectly divisible by 400
    if (year % 400 == 0) {
        r = 1; //printf("%d is a leap year.", year);
    }
    // not a leap year if divisible by 100
    // but not divisible by 400
    else if (year % 100 == 0) {
        r=0; // printf("%d is not a leap year.", year);
    }
    // leap year if not divisible by 100
    // but divisible by 4
    else if (year % 4 == 0) {
        r=1; // printf("%d is a leap year.", year);
    }
    // all other years are not leap years
    else {
        r=0; // printf("%d is not a leap year.", year);
    };

    return r;
}
```

Computing a sequence of dates

- ▶ Computing the number of days from fromdate till todate is following a sequence :

$$d_0 = \text{fromdate} \xrightarrow{\text{nextday}} d_1 \xrightarrow{\text{nextday}} d_2 \dots \xrightarrow{\text{nextday}} d_{n-1} \xrightarrow{\text{nextday}} d_n = \text{todate}$$

- ▶ The value of n is not known and is in fact the number we want to compute.
- ▶ Define a function `nextday` which is returning the next day of a given day.
- ▶ A date is a triple as (d,m,y) and is satisfying the function :

Listing 13 – checkdate(struct date now)

```
#include "structure.h"
#include "leapyear.h"
struct date;
#include "checkdate.h"

int checkdate(struct date now)
{
    int r=0;
    int day=now.day, month=now.month, year=now.year, aleapday;
    int aleapyear=leapyear(year);
    if ( month == 12 || month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month
        if ( 1 <= day && day <= 31) {r=1;};
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        if ( 1 <= day && day <= 30) {r=1;};
    } else if ( aleapyear==1 && month == 2 ) {
        if ( 1 <= day && day <= 29) {r=1;};
    } else if ( aleapyear==0 && month == 2) {
        if ( 1 <= day && day <= 28) {r=1;}; };
    }
    if ( 1 <= day && day <= 28) {r=1;}; };
}
```

Listing 14 – nextday(struct date now)

```
#include "structure.h"
#include "leapyear.h"
#include "checkdate.h"
struct date;
#include "nextday.h"

struct date  nextday(struct date now)
{
    struct date r;
    int day=now.day, month=now.month, year=now.year, aleapday;
    int  aleapyear=leapyear(year);
    int nday=day, nmonth=month, nyear=year;
    if ( month == 12 || month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10
        if (day < 31) {nday=day+1;} else if (day == 31 && month != 12) {nday=1;nmonth=month+1;} else {nday=1;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        if (day < 30) {nday=day+1;} else if (day == 30 ) {nday=1;nmonth=month+1;};
    } else if ( aleapyear==1 && month == 2 && day == 29 ) {nday=1; nmonth=3;}
    else if ( aleapyear==1 && month == 2 && day != 29 ) {nday=day+1;}
    else if ( aleapyear==0 && month == 2 && day == 28 ) {nday=1; nmonth=3;}
    else if ( aleapyear==0 && month == 2 && day != 28 ) {nday=day+1;}
    else {nday=day + 1;}

    r.day = nday;
    r.month = nmonth;
    r.year = nyear;
    return r;
}
```

int daytilldate(struct date currentdate, struct date futuredate)

Listing 15 – int daytilldate(struct date currentdate, struct date futuredate)

```
#include "structure.h"
#include "leapyear.h"
#include "checkdate.h"
#include "nextday.h"
struct date;
#include "daytilldate.h"

int daytilldate(struct date currentdate, struct date futuredate)
{
    int r=0;
    struct date d=currentdate;
    while (d.day != futuredate.day || d.month != futuredate.month || d.year != futuredate.year) {
        r++;
        d = nextday(d);
    };
    return r;
}
```

- ▶ Software Systems assist our daily lifes
- ▶ Questions on dependability and security assurance should be addressed
- ▶ Questions on certification with respect to norms and standards
- ▶ Improving the life-cycle development for addressing these questions

Critical System

Critical systems are systems in which defects could have a dramatic impact on human life or the environment.

Critical System

Critical systems are systems in which defects could have a dramatic impact on human life or the environment.

System failure

Software failure or fault of complex systems is the major cause in the software crisis. For example,

- ▶ **Therac-25 (1985-1987)** : six people overexposed through radiation.
- ▶ **Cardiac Pacemaker (1990-2002)** : 8834 pacemakers were explanted.
- ▶ **Insulin Infusion Pump (IIP) (2010)** : 5000 adverse events.

- ▶ Safety-critical systems : A system whose failure may result in injury, loss of life or serious environmental damage. **An example of a safety-critical system is a control system for a chemical manufacturing plant.**
- ▶ Mission-critical systems : A system whose failure may result in the failure of some goal-directed activity. **An example of a mission-critical system is a navigational system for a spacecraft.**
- ▶ Business-critical systems : A system whose failure may result in very high costs for the business using that system. **An example of a business-critical system is the customer accounting system in a bank.**

Issue

The high costs of failure of critical systems means that trusted methods and techniques must be used for development.

- ▶ Legacy systems are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology.
- ▶ Legacy systems include not only hardware and software but also legacy processes and procedures ;old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components.
- ▶ Legacy systems are often business-critical systems. They are maintained because it is too risky to replace them.
- ▶ For example, for most banks the customer accounting system was one of their earliest systems.

Software Process

A software process is a structured set of activities required to develop a software system : Spiral Model, Waterfall Model, V-Shaped Model, etc.

A software process involve the following steps :

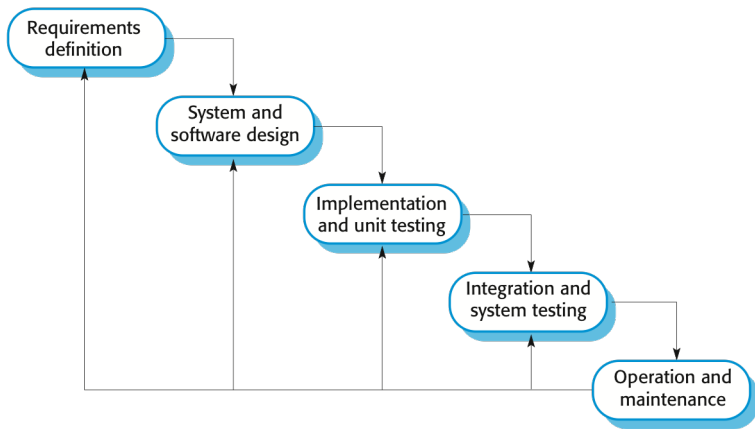
- ▶ Specification *defining what the system should do ;*
- ▶ Design and implementation *defining the organization of the system and implementing the system ;*
- ▶ Validation *checking that it does what the customer wants ;*
- ▶ Evolution *changing the system in response to changing customer needs.*

software process model

A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

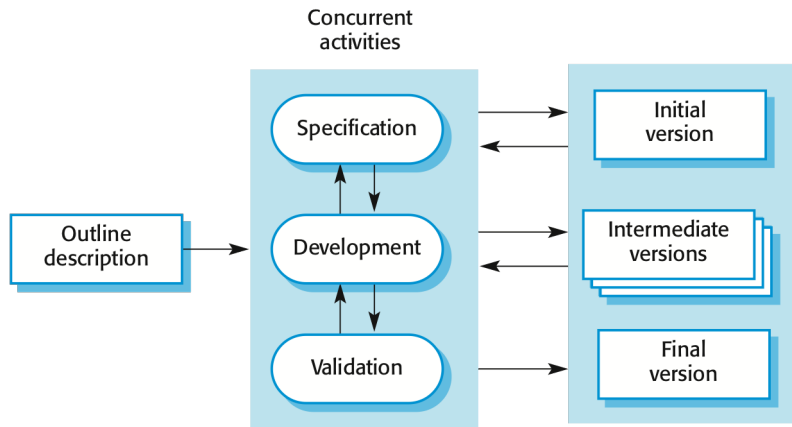
- ▶ The waterfall model : Plan-driven model ; Separate and distinct phases of specification and development.
- ▶ Incremental development : Specification, development and validation are interleaved ; plan-driven or agile.
- ▶ Reuse-oriented software engineering : The system is assembled from existing components ; plan-driven or agile.

The Waterfall Model



(from Software Engineering by Ian Sommerville)

The Incremental Model



(from Software Engineering by Ian Sommerville)

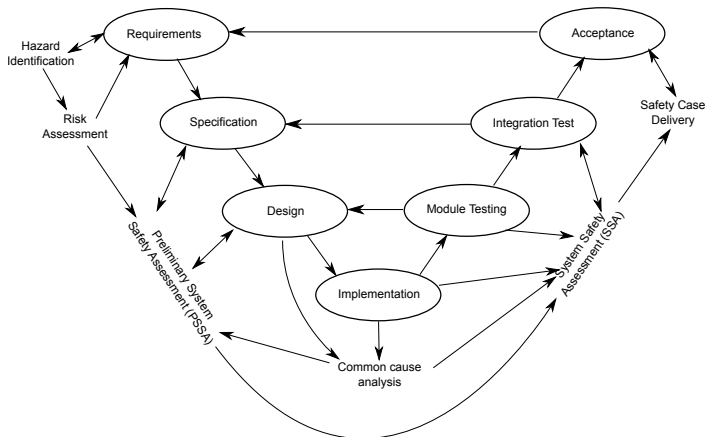


(from Software Engineering by Ian Sommerville)

Boehm's spiral model of the software process



(from Software Engineering by Ian Sommerville)



(from Software Engineering by Ian Sommerville)



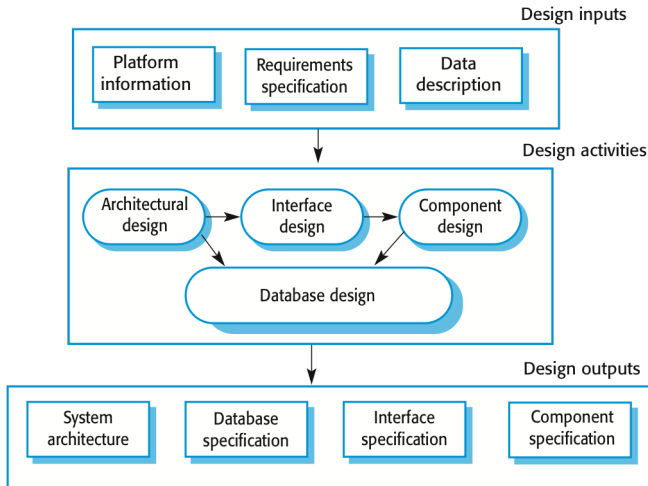
(from Software Engineering by Ian Sommerville)

Requirement Engineering

The process of establishing what services are required and the constraints on the system's operation and development.

- ▶ Feasibility study : Is it technically and financially feasible to build the system ?
- ▶ Requirements elicitation and analysis : What do the system stakeholders require or expect from the system ?
- ▶ Requirements specification : Defining the requirements in detail
- ▶ Requirements validation : Checking the validity of the requirements

General Model of the Design Process



(from Software Engineering by Ian Sommerville)

Testing phases in a plan-driven software process



(from Software Engineering by Ian Sommerville)

SIL

Safety integrity level (SIL) is defined as a relative level of risk-reduction provided by a safety function, or to specify a target level of risk reduction. In simple terms, SIL is a measurement of performance required for a safety instrumented function (SIF).

- ▶ The European functional safety standards based on the IEC 61508 standard defines four SILs (with SIL 4 the most dependable and SIL 1 the least).
- ▶ A SIL is determined based on a number of quantitative factors in combination with qualitative factors such as development process and safety life cycle management.

- ▶ Probability of Failure on Demand during 10 years.
- ▶ SILs :
 - SIL4 : Conséquence très importante sur la communauté entraînant une réduction du danger de 10 000 à 100 000.
 - SIL3 : Conséquence très importante sur la communauté et les employés entraînant une réduction du danger de 1 000 à 10 000.
 - SIL2 : Protection importante de l'installation, de la production et des employés entraînant une réduction du danger de 100 à 1000.
 - SIL1 : Faible protection de l'installation, de la production entraînant une réduction du danger de 10 à 100.

Grâce à une grande maîtrise en calcul formel, en sécurité de fonctionnement et à l'utilisation de la méthode B (beaucoup utilisée en milieu industriel pour réaliser des logiciels sécuritaires prouvés), ClearSy System Engineering est qualifiée pour mener à bien des projets nécessitant un contexte de certification de niveau SIL 2, SIL 3, ou SIL 4, selon la norme 61508.

- ▶ Validation : *Are we building the right product*
- ▶ Verification : *Are we building the process right ?*

verification

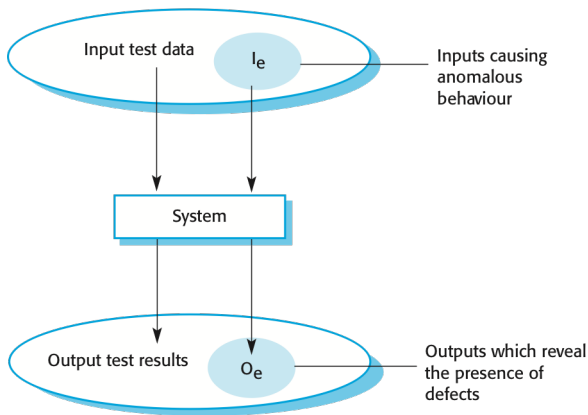
The verification aims to check that the software meets its stated functional and non-functional requirements.

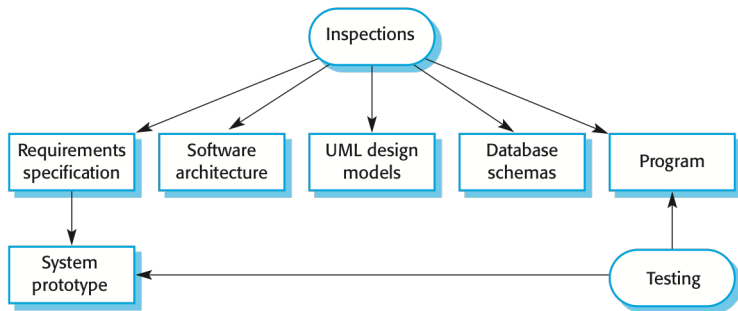
- ▶ *functional requirements*
- ▶ *non-functional requirements*

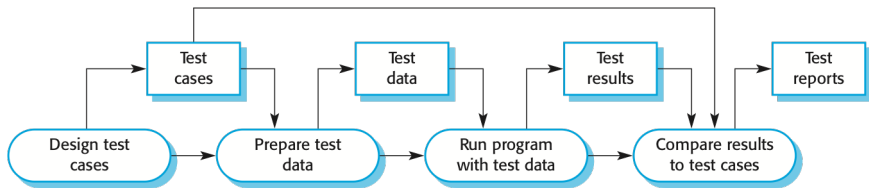
validation

The verification aims to ensure that the software meets the customer's expectations.

An input-output model of program testing







- ▶ Validation : *Are we building the right product*
- ▶ Verification : *Are we building the process right ?*

verification

The verification aims to check that the software meets its stated functional and non-functional requirements.

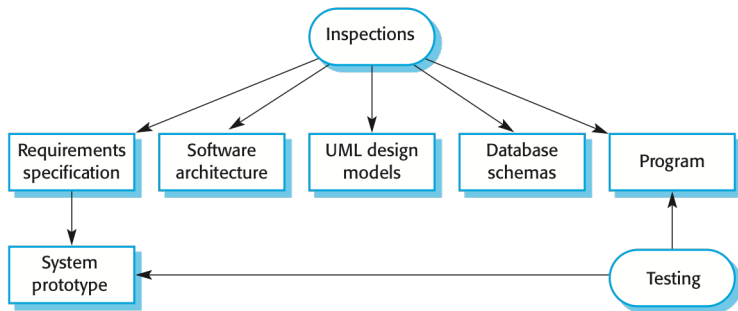
- ▶ *functional requirements*
- ▶ *non-functional requirements*

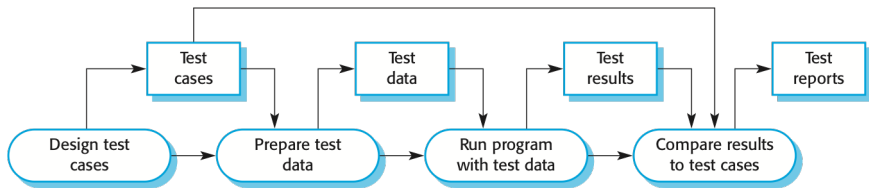
validation

The verification aims to ensure that the software meets the customer's expectations.

An input-output model of program testing







- ▶ The Cleanroom method, developed by the late Harlan Mills and his colleagues at IBM and elsewhere, attempts to do for software what cleanroom fabrication does for semiconductors : to achieve quality by keeping defects out during fabrication.
- ▶ In semiconductors, dirt or dust that is allowed to contaminate a chip as it is being made cannot possibly be removed later.
- ▶ But we try to do the equivalent when we write programs that are full of bugs, and then attempt to remove them all using debugging.

The Cleanroom method, then, uses a number of techniques to develop software carefully, in a well-controlled way, so as to avoid or eliminate as many defects as possible before the software is ever executed. Elements of the method are :

- ▶ specification of all components of the software at all levels ;
- ▶ stepwise refinement using constructs called "box structures" ;
- ▶ verification of all components by the development team ;
- ▶ statistical quality control by independent certification testing ;
- ▶ no unit testing, no execution at all prior to certification testing.

Software development in five key strategies

- ▶ **Formal specification** : The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
- ▶ **Incremental development** : The software is partitioned into increments which are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.
- ▶ **Structured programming** : Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to apply correctness-preserving transformations to the specification to create the program code.
- ▶ **Static verification** : The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- ▶ **Statistical testing of the system** : The integrated software increment is tested statistically, to determine its reliability. These statistical tests are based on an operational profile which is developed in parallel with the system specification.

Diagram for the Cleanroom Model





- ▶ Informal Requirements : Restricted form of natural language.
- ▶ Formal Specification : Modeling language like Event-B , Z, ASM, VDM, TLA+...
- ▶ Formal Verification : Theorem Prover Tools like PVS, Z3, SAT, SMT Solver...
- ▶ Formal Validation : Model Checker Tools like ProB, UPPAAL , SPIN, SMV ...
- ▶ Real-time Animation : **Our proposed approach ... Real-Time Animator ...**
- ▶ Code Generation : **Our proposed approach ... EB2ALL : EB2C, EB2C++, EB2J, EB2C# ...**
- ▶ Acceptance Testing : Failure Mode, Effects and Critically analysis(FMEA and FMEA), System Hazard Analyses(SHA)

- ▶ Typing Properties using Typechecker (see for instance functional programming languages as ML, CAML, OCAML, ...)
- ▶ Invariance and safety (*A nothing bad will happen !*) properties for a program P :
 - Transformation of P into a relational model M simulating P
 - Expression of safety properties :
$$\forall s, s' \in \Sigma. (s \in \text{Init}_S \wedge s \xrightarrow{*} s') \Rightarrow (s' \in A).$$
 - Definition of the set of reachable states of P using M :
$$\text{REACHABLE}(M) = \text{Init}_S \cup \longrightarrow [\text{REACHABLE}(M)]$$
 - Main property of $\text{REACHABLE}(M)$: $\text{REACHABLE}(M) \subseteq A$
 - Characterization of $\text{REACHABLE}(M)$:
$$\text{REACHABLE}(M) = \text{FP}(\text{REACHABLE}(M))$$

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$:

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically!

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically!
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$:

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically!
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically!
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$:

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically!
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible for some classes of systems and with some tools.

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically !
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible for some classes of systems and with some tools.
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: changing the domain and solving in another domain as abstract interpretation if making possible

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically !
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible for some classes of systems and with some tools.
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: changing the domain and solving in another domain as abstract interpretation if making possible
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$:

- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: undecidable ... no program is able to prove it automatically !
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible when restrictions over the set of states is possible (finite set of states)
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: possible for some classes of systems and with some tools.
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: changing the domain and solving in another domain as abstract interpretation if making possible
- ▶ Proving automatically $\text{REACHABLE}(M) \subseteq A$: approximating semantics of programs

Decidability or Undecidability

- A problem $x \in P$ is generally stated by the function $\chi_{x \in P}$ where $\chi_{x \in P}(u) = 1$, if $P(u)$ is true and $\chi_{x \in P}(u) = 0$, if $P(u)$ is false :
- Problem 1 : $x \in 0..n$ where $n \in \mathbb{N}$
 - Problem 1 : $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^*$.

- ▶ A problem $x \in P$ is generally stated by the function $\chi_{x \in P}$ where $\chi_{x \in P}(u) = 1$, if $P(u)$ is true and $\chi_{x \in P}(u) = 0$, if $P(u)$ is false :
 - Problem 1 : $x \in 0..n$ where $n \in \mathbb{N}$
 - Problem 1 : $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^*$.
- ▶ A problem $x \in P$ is decidable, when the function $\chi_{x \in P}$ is computable or more precisely the function can be computed by a program

- ▶ A problem $x \in P$ is generally stated by the function $\chi_{x \in P}$ where $\chi_{x \in P}(u) = 1$, if $P(u)$ is true and $\chi_{x \in P}(u) = 0$, if $P(u)$ is false :
 - Problem 1 : $x \in 0..n$ where $n \in \mathbb{N}$
 - Problem 1 : $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^*$.
- ▶ A problem $x \in P$ is decidable, when the function $\chi_{x \in P}$ is computable or more precisely the function can be computed by a program
- ▶ Problem of the correctness of a program :
 - Assume that \mathcal{F} is the set of unary function over natural numbers : $\mathcal{F} = \mathbb{N} \rightarrow \mathbb{N}$.
 - $\mathcal{C} \subseteq \mathcal{F}$: the set of computable (or programmable) functions is \mathcal{C}
 - $f \in \mathcal{C} = \{\Phi_0, \Phi_1, \dots, \Phi_n, \dots\}$: the set of computable functions is denumerable.
 - The problem $x \in \text{dom}(\Phi_y)$ is not decidable and it expresses the correctness of programs.

Implicite versus explicite

- Ecrire $101 = 5$ peut avoir une signification

Implicite versus explicite

- ▶ Ecrire $101 = 5$ peut avoir une signification
- ▶ Le code du nombre n est 101 à gauche du symbole $=$ et le code du nombre n est sa représentation en base 10 à droite.
- ▶ $n_{10} = 5$ et $n_2 = 101$
- ▶ Vérification : $base(2, 10, 101) = 1.2^2 + 0.2 + 1.2^0 = 5_{10}$

Example : description of static behaviour

- ▶ A train moving at absolute speed $spd1$
- ▶ A person walking in this train with relative speed $spd2$
 - One may compute the absolute speed of the person
- ▶ Modelling
 - Syntax. Classical expressions
 - ▶ Type $Speed = Float$
 - ▶ $spd1, spd2 : Speed$
 - ▶ $AbsoluteSpeed = spd1 + spd2$
 - Semantics
 - ▶ If $spd1 = 25.6$ and $spd2 = 24.4$ then $AbsoluteSpeed = 50.0$
 - ▶ If $spd1 = "val"$ and $spd2 = 24.4$ then exception raised
 - Pragmatics
 - ▶ What if $spd1$ is given in *mph* (miles per hour) and $spd2$ in *km/s* (kilometers per second) ?
 - ▶ What if $spd1$ is a relative speed ?

- ▶ Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
 - $STATES$ est l'ensemble de tous les états de P : $STATES = X \rightarrow \mathbb{Z}$ où X désigne les variables de P .
 - s_0 et s_f deux états de $STATES$: $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f .
 - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
- STATES est l'ensemble de tous les états de P : $STATES = X \rightarrow \mathbb{Z}$ où X désigne les variables de P .
 - s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f .
 - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :

$$s_0(X) = x_0, s_f(X) = x_f, s'(X) = x'$$

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :

- $STATES$ est l'ensemble de tous les états de P : $STATES = X \rightarrow \mathbb{Z}$ où X désigne les variables de P .
- s_0 et s_f deux états de $STATES$: $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f .
- Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :

$$s_0(X) = x_0, s_f(X) = x_f, s'(X) = x'$$

- $\mathcal{D}(P)(s_0) = s_f$ définit la relation suivante sur l'ensemble des valeurs :

$$x_0 \xrightarrow{P} x_f$$

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :

- STATES est l'ensemble de tous les états de P : $STATES = X \rightarrow \mathbb{Z}$ où X désigne les variables de P .
- s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f .
- Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :

$$s_0(X) = x_0, s_f(X) = x_f, s'(X) = x'$$

- $\mathcal{D}(P)(s_0) = s_f$ définit la relation suivante sur l'ensemble des valeurs :

$$x_0 \xrightarrow{P} x_f$$

- Un programme P *remplit* un contrat (pre,post) :

- P transforme une variable x à partir d'une valeur initiale x_0 et produisant une valeur finale x_f : $x_0 \xrightarrow{P} x_f$
- x_0 satisfait pre : $\text{pre}(x_0)$
- x_f satisfait post : $\text{post}(x_0, x_f)$
- $\text{pre}(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow \text{post}(x_0, x_f)$







- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*



- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $C(S) \vdash P \Rightarrow WP(S)(Q)$: *logical formula*



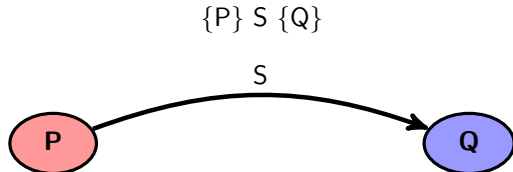
- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $C(S) \vdash P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $SP(S)(P) \Rightarrow Q$: *logical formula*



- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $C(S) \vdash P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $SP(S)(P) \Rightarrow Q$: *logical formula*
- ▶ $C(S) \vdash SP(S)(P) \Rightarrow Q$: *logical formula*



- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $C(S) \vdash P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $SP(S)(P) \Rightarrow Q$: *logical formula*
- ▶ $C(S) \vdash SP(S)(P) \Rightarrow Q$: *logical formula*



- ▶ $\{P\} S \{Q\}$: *asserted program*
- ▶ $P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $C(S) \vdash P \Rightarrow WP(S)(Q)$: *logical formula*
- ▶ $SP(S)(P) \Rightarrow Q$: *logical formula*
- ▶ $C(S) \vdash SP(S)(P) \Rightarrow Q$: *logical formula*

Predicate Transformer

$WP(S)(Q)$ is the Weakest-Precondition of S for Q and is a predicate transformer but $WP(S)(.)$ is not a computable function over the set of predicates.

Summary

Modelling, specification and verification

Modelling, specification and verification

- ▶ Set-theoretical notations using the Event-B modelling language
- ▶ Relational modelling of a program or an algorithm
- ▶ Program properties as safety, invariance, pre and post conditions
- ▶ Design by contract
- ▶ Method for proving invariance properties of programs and induction principles as Floyd's method, Hoar logics,
- ▶ Techniques for Model-Checking
- ▶ Tools : the toolset RODIN, the toolset TLAPS, the toolset PAT, the toolset Eiffel Studio, the toolset Frama-c, ...

Summary

Logics

Logics

- ▶ Propositional Formulae and first order formulae
- ▶ Models
- ▶ Sequents Calculus
- ▶ Proofs and deduction
- ▶ Resolution
- ▶ Tools : the toolset Rodin

Summary

Fixed-Point Theory

Fixed-Point Theory

- ▶ Complete Partially Ordered Sets (CPO) and Complete Lattices
- ▶ Fixed-Point Theorems : Kleene, Tarski, ...
- ▶ Abstract Interpretation
- ▶ Galois Lattices
- ▶ Static analysis of Programs by abstract interpretation.
- ▶ Semantics of programs

Summary

Computability, Decidability, complexity, Undecidability

Computability, Decidability, complexity, Undecidability

- ▶ Models of computing : Turing Machines, Partially Recursive Functions, URM, ...
- ▶ Church's Thesis
- ▶ Decidability
- ▶ Undecidability
- ▶ Complexity

Summary of concepts



Summary

Tools

Tools

- ▶ The TLA⁺ ToolBox
- ▶ The RODIN platform
- ▶ Frama-C
- ▶ Boogie and the Visual Studio Suite
- ▶ UPPAAL

- ▶ Documents sur Arche MODÈLES ET ALGORITHMES avec le mot de passe *mery2023*
- ▶ Alternance des cours et des TDs avec des séances sur machines.
- ▶ Intervention de Rosemary Monahan de NUI Maynooth en cours d'année pour un cours et un TD dupliqué pour IL.
- ▶ Deux groupes de TD
- ▶ Machine virtuelle et machines telecom avec les logiciels installés.