

---

# Cours MVSI

## Modélisation et Vérification des Systèmes Informatiques

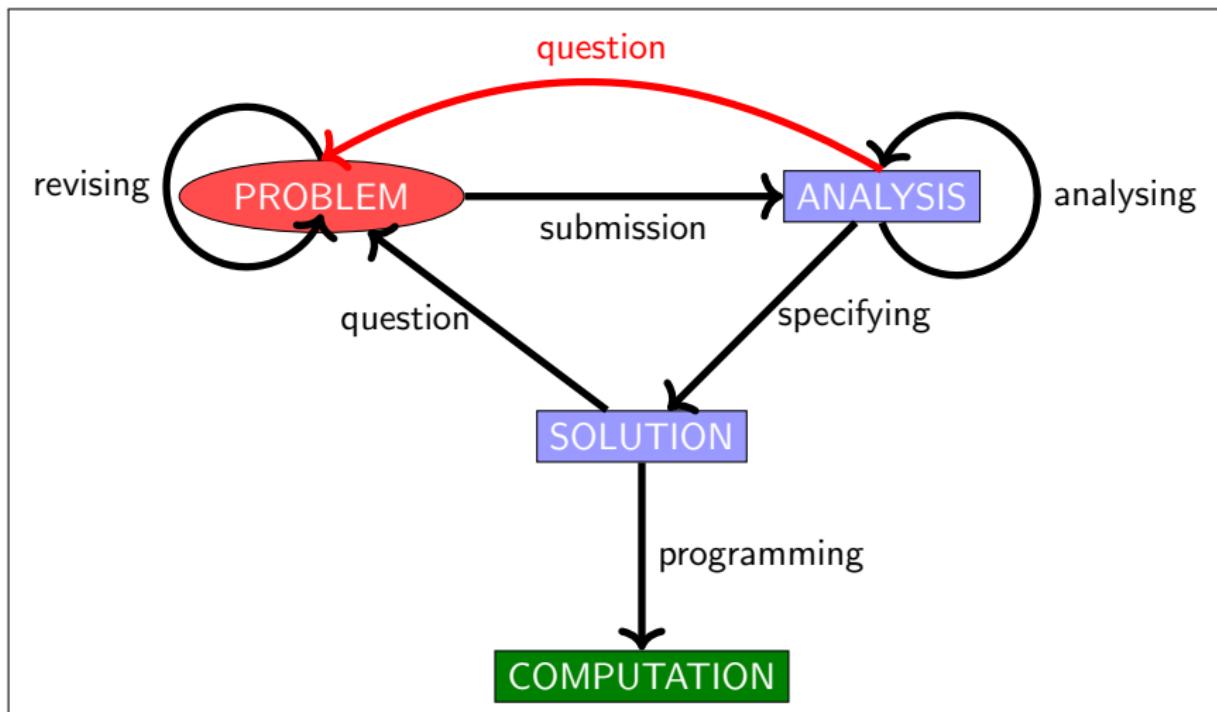
### Overview of the course

---

Dominique Méry  
Telecom Nancy, Université de Lorraine  
(17 novembre 2025 at 12:17 A.M.)

- ① Tracking bugs in C codes
- ② Introduction by Example
  - Detecting overflows in computations
  - Computing the velocity of an aircraft on the ground
- ③ Verification of program properties
- ④ Topics of course

## Problem versus Solution



## 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 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;
}
```

**bug0.c prints Result : w**

## Listing 3 – 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--and-y=%d\n", i, y);
    }

    return 0;
}
```

## Listing 4 – 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--and-y=%d\n", i, y);
    }

    return 0;
}
```

**bug00.c prints ... Result : x= w1 ;  
Result : i=100000 w2**

## Listing 5 – Bug bug000

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

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

    for (i = 0; i <= 200000; 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--%d\n", i, y);
    }

    return 0;
}
```

## Listing 6 – Bug bug000

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

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

    for (i = 0; i <= 200000; 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--%d\n", i, y);
    }

    return 0;
}
```

Result: x= 70

Result: i=200000 2

## Listing 7 – Bug bug1

```
#include <stdio.h>

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int sum = 0;

    // Attempt to calculate the sum of numbers in the array
    for (int i = 0; i <= 5; i++) {
        sum += numbers[i];
    }

    printf("Sum: -%d\n", sum);
    sum = numbers[0];
    printf("Sum: -%d\n", sum);

    return 0;
}
```

## Listing 8 – Bug bug1

```
#include <stdio.h>

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int sum = 0;

    // Attempt to calculate the sum of numbers in the array
    for (int i = 0; i <= 5; i++) {
        sum += numbers[i];
    }

    printf("Sum: -%d\n", sum);
    sum = numbers[0];
    printf("Sum: -%d\n", sum);

    return 0;
}
```

Sum: 16

Sum: 1

## Listing 9 – 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 10 – 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;
}
```

Infinite loop . . .

## Listing 11 – Bug bug7

```
#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;
}
```

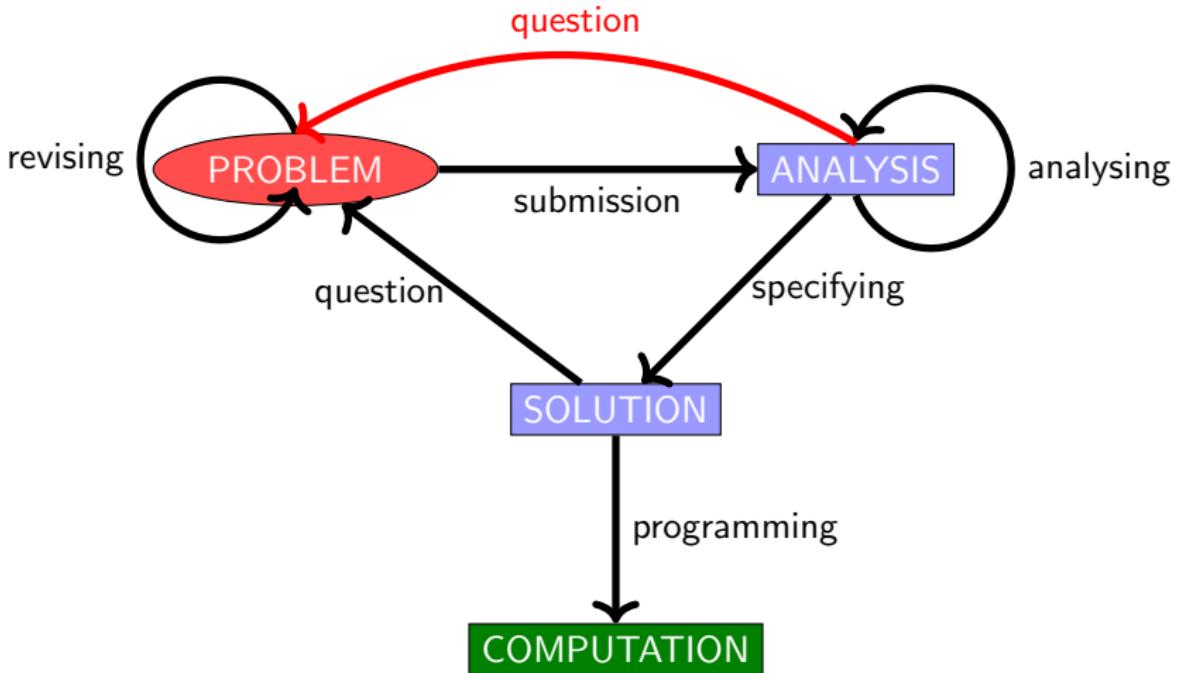
## Listing 12 – Bug bug7

```
#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;
}
```

Average for 2147483647 and  
2147483647 is -1

## Problem versus Solution



Listing 13 – 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;
}
```

## Annotated Example 1

---

Listing 14 – 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);
}
```

## Nose Gear Velocity

---



- ▶ 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

► 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).

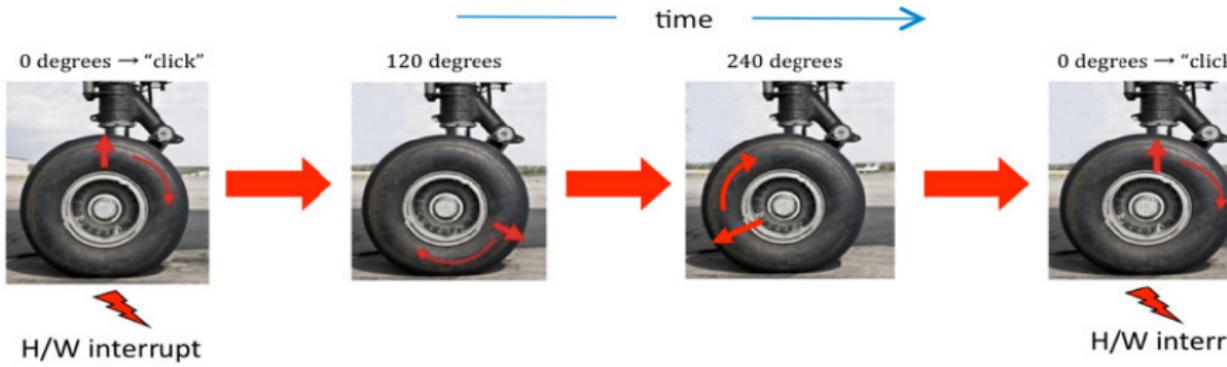
What are the main properties to consider for formalization ?

- ▶ 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  $\text{estimateVelocity} \in \text{MILES} \times \text{HOURS} \rightarrow \text{MPH}$
  - 2  $\text{mphToKph} \in \text{MPH} \rightarrow \text{KPH}$

## Sample Velocity Estimation



NGClickTime = 4123 millisecs  
NGRotations = 8954

NGClickTime = 4367 millisecs  
NGRotations = 8955

WHEEL\_DIAMETER = 22 inches  
PI = 3.14

12 inches/foot  
5280 feet/mile

$$\begin{aligned}\text{estimatedGroundVelocity} &= \text{distance travel/elapsed time} \\ &= ((3.14 * 22)/(12*5280))/((4367-4123)/(1000*3600)) \\ &= 16 \text{ mph}\end{aligned}$$

### Safety Property

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

- ▶  $\text{Length} = \pi \cdot \text{diameter} \cdot \text{VNGClick}$  (mathematical property)
- ▶  $\text{Length} \leq 6000$  (domain property)
- ▶  $\pi \cdot \text{diameter} \cdot \text{VNGClick} \leq 6000$
- ▶  $\text{VNGClick} \leq 6000 / (\pi \cdot \text{diameter})$
- ▶ if  $n=8$ , then  $2^7 - 1 = 127$  and  
 $6000 / (\pi \cdot [22, \text{inch}]) = 6000 / (\pi \cdot 55, 88) = 6000 / (3, 24 \cdot [55, 88, \text{cm}]) = 6000 / (3, 24 \cdot 0.5588) \approx 3419$  and the condition of safety can not be satisfied in any situation.
- ▶ if  $n=16$ , then  $2^{15} - 1 = 65535$  and  $6000 / (\pi \cdot [22, \text{inch}]) \approx 3419$  and the condition of safety can be satisfied in any situation since  $3419 \leq 65535$

### Safety Property

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

$$\text{RTE\_VNGClick} : 0 \leq vNGClick \leq \text{INT\_MAX} \quad (1)$$

- ▶ The current value of VNGClick is always bounded by the two values 0 and INT\_MAX.

- ▶ 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 validation aims to ensure that the software meets the customer's expectations.

- ▶ 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 :
$$\forall s, s' \in \Sigma. (s \in \text{Inits} \wedge s \xrightarrow{*} s') \Rightarrow (s' \in A).$$
  - Definition of the set of reachable states of P using M :
$$\text{REACHABLE}(M) = \text{Inits} \cup \xrightarrow{} [\text{REACHABLE}(M)]$$
  - 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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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 :  $base(2, 10, 101) = 1 \cdot 2^2 + 0 \cdot 2 + 1 \cdot 2^0 = 5_{10}$

- ▶ 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 :  $pre(x_0)$
- $x_f$  satisfait post :  $post(x_0, x_f)$
- $pre(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow post(x_0, x_f)$

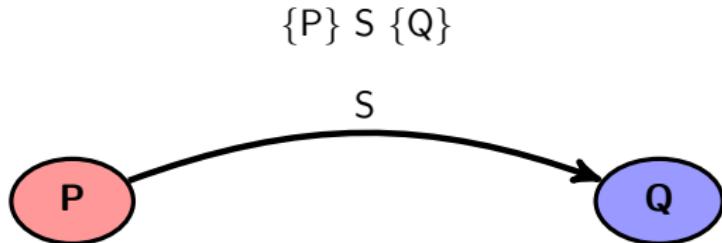
Un programme P *remplit* un contrat (pre,post) :

- ▶ 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 :  $\text{pre}(x_0)$  and  $x_f$  satisfait post :  $\text{post}(x_0, x_f)$
- ▶  $\text{pre}(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow \text{post}(x_0, x_f)$

```
requires pre(x0)
ensures post(x0, xf)
variables X
```

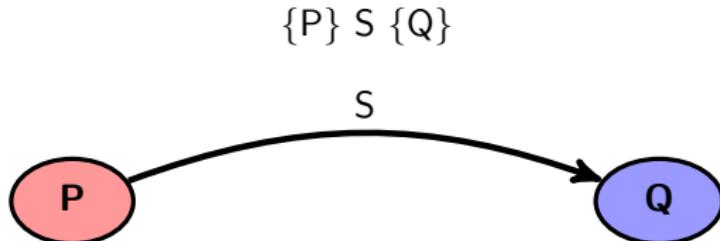
```
begin
  0 : P0(x0, x)
  instruction0
  ...
  i : Pi(x0, x)
  ...
  instructionf-1
  f : Pf(x0, x)
end
```

- ▶  $\text{pre}(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- ▶  $P_f(x_0, x) \Rightarrow \text{post}(x_0, x)$
- ▶ some conditions for verification related to pairs  $\ell \longrightarrow \ell'$



## Asserted Program $\{P\} S \{Q\}$

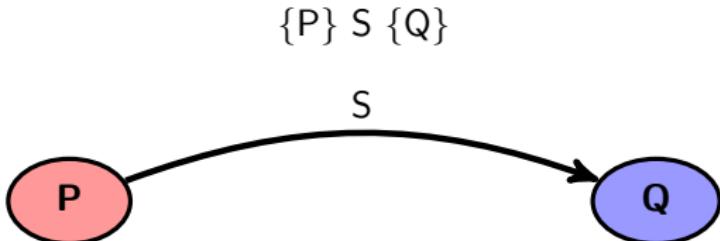
---



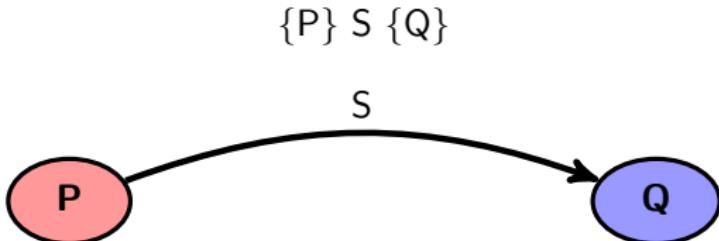
- ▶  $\{P\} S \{Q\}$  : asserted program

## Asserted Program $\{P\} S \{Q\}$

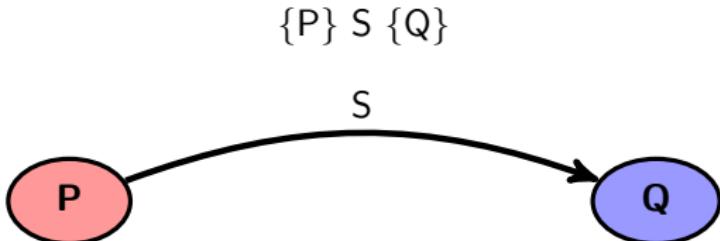
---



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



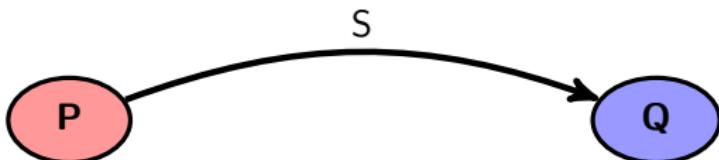
- ▶  $\{P\} S \{Q\}$  : asserted program
- ▶  $P \Rightarrow WP(S)(Q)$  : logical formula
- ▶  $C(S) \vdash P \Rightarrow WP(S)(Q)$  : logical formula



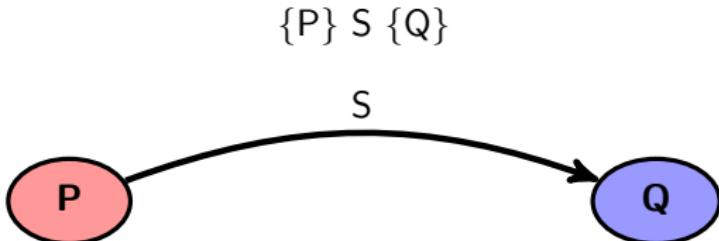
- ▶  $\{P\} S \{Q\}$  : *asserted program*
- ▶  $P \Rightarrow WP(S)(Q)$  : *logical formula*
- ▶  $C(S) \vdash P \Rightarrow WP(S)(Q)$  : *logical formula*
- ▶  $SP(S)(P) \Rightarrow Q$  : *logical formula*

## Asserted Program {P} S {Q}

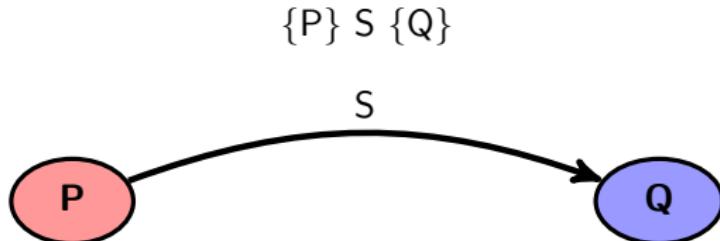
$\{P\} \leq \{Q\}$



- ▶  $\{P\} S \{Q\}$  : 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.

- ▶ Découpage de l'unité : 10 cours de 2 h

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ⌚ Principes de modélisation des systèmes informatiques : systèmes de transition

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils
  - ☺ TLA/TLA<sup>+</sup> avec TLC,

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils
  - ☺ TLA/TLA<sup>+</sup> avec TLC, Rodin,

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils
  - ☺ TLA/TLA<sup>+</sup> avec TLC, Rodin, Frama-c

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils
  - ☺ TLA/TLA<sup>+</sup> avec TLC, Rodin, Frama-c
  - PAT, Spec#, DAFNY, Spin/Promela, Z3, Kind2, Infer  
<https://fbinfer.com>

- ▶ Découpage de l'unité : 10 cours de 2 h
- ▶ Contenu :
  - ☺ Principes de modélisation des systèmes informatiques : systèmes de transition
  - ☺ Propriétés d'un système informatique : sûreté, vivacité, disponibilité, sécurité, dépendabilité
  - ☺ Modélisation de propriétés de systèmes
  - ☺ Analyse des programmes
  - ☺ Vérification de propriétés de systèmes avec un model checker TLC
  - ☺ Vérification de propriétés de systèmes avec un outil de preuve Rodin
  - ☺ Vérification de propriétés de systèmes avec un analyseur Frama-c
- ▶ Outils
  - ☺ TLA/TLA<sup>+</sup> avec TLC, Rodin, Frama-c
  - PAT, Spec#, DAFNY, Spin/Promela, Z3, Kind2, Infer  
<https://fbinfer.com>
- ▶ Contrôle des connaissances : deux écrits et un TP
  - ☺ Ecrit et TP