MVSI Course
Modelling and Verification of Computer Systems

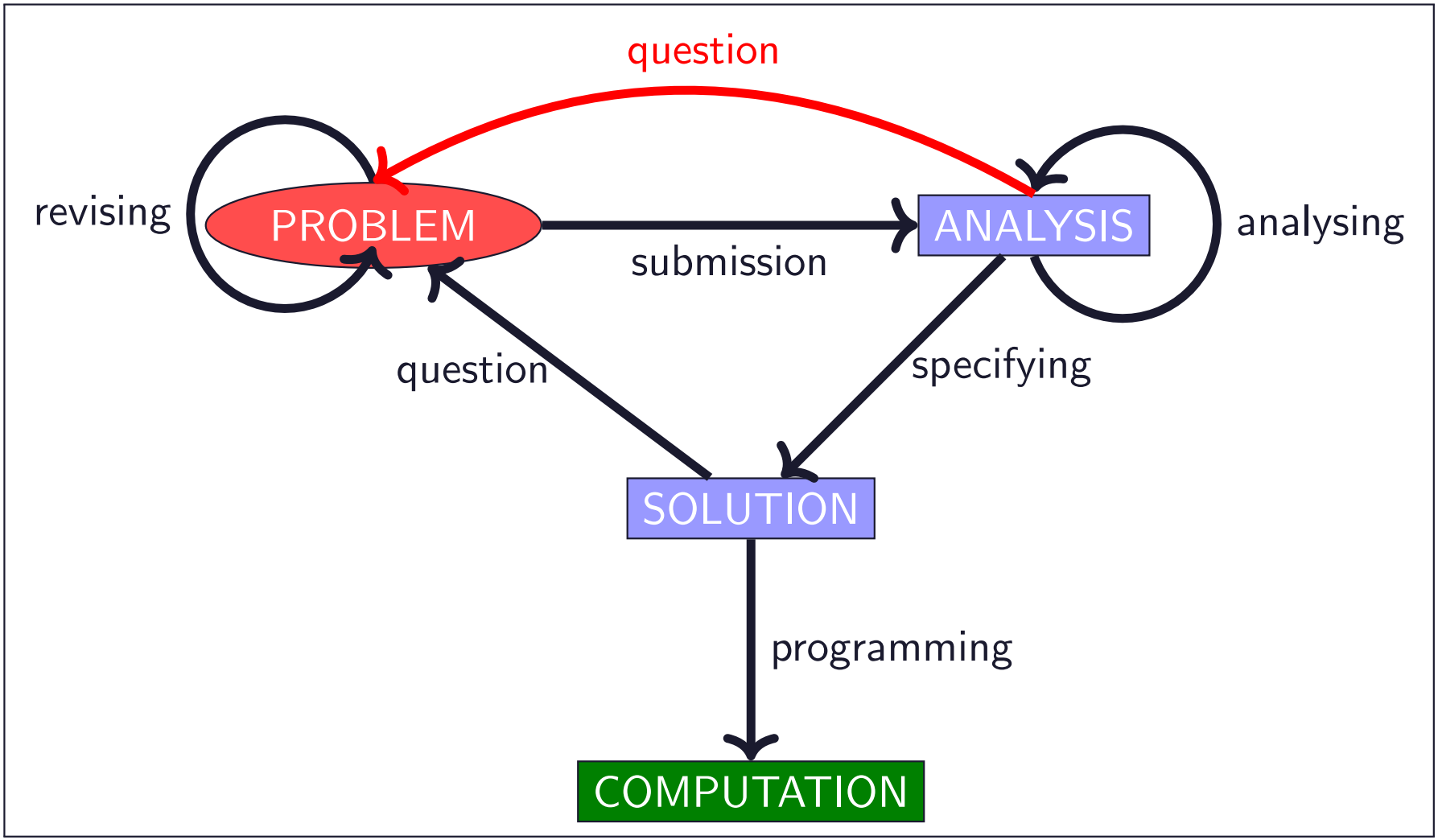
Lecture 0 – Course Overview

Dominique Méry
Telecom Nancy, Université de Lorraine
November 20, 2026

Plan

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 the course

Problem versus Solution



Part 1

Tracking bugs in C codes

bug0.c

Listing 1: Bug bug0

```c
#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("x=%d␣␣and␣y=%d\n",x, y);
    return 0;
}
```

bug0.c

Listing 2: Bug bug0

```c
#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("x=%d␣␣and␣y=%d\n",x, y);
    return 0;
}
```

bug0.c prints x=some number and y=some other number

bug00.c

Listing 3: Bug bug00

```
1  // Heisenbug
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  //
6  int main() {
7    int x, y,i=0;
8      for (i = 0; i <= 1000000; i++) {
9      // Seed the random number generator with the current time
10     srand(time(NULL));
11     // Generate a random number between 1 and 100
12     x = rand() % 100 + 1;
13         printf("Result: x=  %d\n",x);
14     // Perform some calculations
15     y = x / (100 - x);
16     printf("Result: i=%d  and y=%d\n",i, y);
17     }
18     return 0;
19 }
```

bug00.c

Listing 4: Bug bug00

```
1  // Heisenbug
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  //
6  int main() {
7    int x, y,i=0;
8      for (i = 0; i <= 1000000; i++) {
9      // Seed the random number generator with the current time
10     srand(time(NULL));
11     // Generate a random number between 1 and 100
12     x = rand() % 100 + 1;
13         printf("Result: x=  %d\n",x);
14     // Perform some calculations
15     y = x / (100 - x);
16     printf("Result: i=%d  and y=%d\n",i, y);
17     }
18     return 0;
19 }
```

bug00.c prints ...i=some number and j=some other number ...

bug000.c

Listing 5: Bug bug000

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



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

    for (i = 0; i <= INT_MAX; 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 x = %d  and y=%d\n",i,x, y);
    }

    return 0;
```

bug000.c

Listing 6: Bug bug000

```
1  // Heisenbug
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <limits.h>
6
7
8
9  int main() {
10   int x, y,i=0;
11
12     for (i = 0; i <= INT_MAX; i++) {
13     // Seed the random number generator with the current time
14     srand(time(NULL)+i);
15
16     // Generate a random number between 1 and 100
17      x = rand() % 100 + 1;
18
19         printf("Result: x=  %d\n",x);
20     // Perform some calculations
21     y = x / (100 - x);
22
23     printf("Result: i=%d x = %d  and y=%d\n",i,x, y);
24     }
25
26     return 0;
```

RTE

- RTE means Run Time Error
- `x = rand() % 100 + 1` and `y = x / (100 - x)`
- $rte(\texttt{x = rand() \% 100 + 1}) \stackrel{def}{=}$
  $rte(\texttt{x}) \wedge rte(\texttt{rand()}) \wedge rte(\texttt{rand()\%100+1})$
- $rte(\texttt{y = x / (100 - x)}) \stackrel{def}{=} rte(\texttt{y}) \wedge rte(\texttt{x}) \wedge 100 \neq x$

RTE

The value of `x` may be *100*, and a sequence of tests may never produce `x` with the value 100.

bug1.c

Listing 7: Bug bug1

```c
#include <stdio.h>
#define n 5
int main() {
    int numbers[n] = {1, 2, 3, 4, 5};
    int sum = 0;

    // Attempt to calculate the sum of numbers in the array
    for (int i = 0; i <= n; i++) {
        sum += numbers[i];
    };
    int val = n*(n+1)/2;
    printf("Iteratively computed sum=%d and direct comutation val=%d
        \n", sum,val);
    return 0;
}
```

bug1.c

Listing 8: Bug bug1

```
1  #include <stdio.h>
2  #define n 5
3  int main() {
4      int numbers[n] = {1, 2, 3, 4, 5};
5      int sum = 0;
6
7      // Attempt to calculate the sum of numbers in the array
8      for (int i = 0; i <= n; i++) {
9          sum += numbers[i];
10     };
11     int val = n*(n+1)/2;
12     printf("Iteratively␣computed␣sum=%d␣and␣direct␣comutation␣val=%d
           \n", sum,val);
13     return 0;
14 }
```

```
Iteratively computed sum=16 and
direct computation val=15
```

bug2.c

Listing 9: Bug bug2

```c
#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;
}
```

bug2.c

Listing 10: Bug bug2

```c
#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 . . .

bug7.c

Listing 11: Bug bug7

```
1  #include <stdio.h>
2  #include <limits.h>
3  int average(int a,int b)
4  {
5    return((a+b)/2);
6  }
7
8  int main()
9  {
10   int x,y;
11   x=INT_MAX;y=INT_MAX;
12   printf("Average␣␣for␣%d␣and␣%d␣is␣%d\n",x,y,
13          average(x,y));
14   return 0;
15 }
```

bug7.c

Listing 12: Bug bug7

```
1  #include <stdio.h>
2  #include <limits.h>
3  int average(int a,int b)
4  {
5    return((a+b)/2);
6  }
7
8  int main()
9  {
10   int x,y;
11   x=INT_MAX;y=INT_MAX;
12   printf("Average␣␣for␣%d␣and␣%d␣is␣%d\n",x,y,
13          average(x,y));
14   return 0;
15 }
```

```
Average for 2147483647 and
2147483647 is -1
```

Part 2

Introduction by Example

Detecting overflows of computations

Listing 13: Function average

```
1 #include <stdio.h>
2 #include <limits.h>
3 int average(int a,int b)
4 {
5   return((a+b)/2);
6 }
```

Execution

Execution produces a result

```
Average  for 2147483647 and 2147483647 is -1
```

Execution

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, annotated

```
1  #include <stdio.h>
2  #include <limits.h>
3  /*@  requires  0 <= a;
4       requires a <= INT_MAX ;
5       requires  0 <= b;
6       requires b <= INT_MAX ;
7       requires  0 <= a+b;
8       requires a+b <= INT_MAX ;
9       ensures \result <= INT_MAX;
10 */
11 int average(int a,int b)
12 {
13   return((a+b)/2);
14 }
15
16 int main()
17 {
18   int x,y;
19   x=INT_MAX / 2;y=INT_MAX / 2;
20   //  printf("Average  for %d and %d is %d\n",x,y,
21   //      );
22   return average(x,y);
23 }
```

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

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: the functional requirement is expressed in *SI* unit system (km/hr).

Characterization of a System (II) cont.

What are the main properties to consider for formalization?

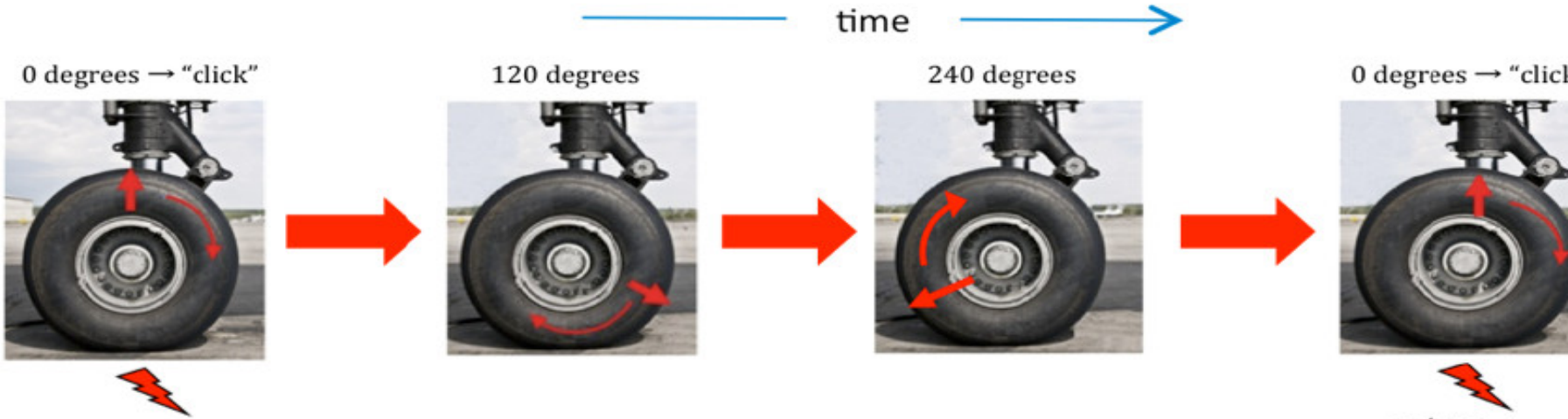
- Two different types of data:
  - counters with modulo semantics
  - non-negative values for time, distance, and velocity
- Two dimensions: *distance* and *time*
- Many units: distance (`inches`, `kilometers`, `miles`), time (`milliseconds`, `hours`), velocity (`kph`, `mph`)
- And interaction among 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 $estimateVelocity \in \texttt{MILES} \times \texttt{HOURS} \rightarrow \texttt{MPH}$
2 $mphTokph \in \texttt{MPH} \rightarrowtail\!\!\!\!\!\rightarrow \texttt{KPH}$

Sample Velocity Estimation



NGClickTime = 4123 millisecs
NGRotations = 8954

NGClickTime = 4367milli
NGRotations = 8955

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 Run Time Error (RTE)

Safety Property

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

- $Length = \pi{\cdot}diameter{\cdot}VNGClick$ (mathematical property)
- $Length \leq 6000$ (domain property)
- $\pi{\cdot}diameter{\cdot}VNGClick \leq 6000$
- $VNGClick \leq 6000/(\pi{\cdot}diameter)$
- if $n = 8$, then $2^7{-}1 = 127$ and $6000/(\pi{\times}[22\ inch]) = 6000/(\pi{\times}[55.88\ cm]) = 6000/(3.14{\times}0.5588) \approx 3419$: the safety condition cannot be guaranteed.
- if $n = 16$, then $2^{15}{-}1 = 32767$ and $6000/(\pi{\times}[22\ inch]) \approx 3419$: the safety condition is satisfied in every situation, since $3419 \leq 32767$.

Safety Property Run Time Error (RTE)

Safety Property

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

$$RTE_VNGClick : 0 \leq VNGClick \leq INT_MAX \tag{1}$$

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

Verification and Validation

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

Verification

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

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

Validation

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

Part 3

Verification of program properties

Verification of program properties

- 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:
    $\forall s, s' \in \Sigma.(s \in Init_{\mathrm{S}} \wedge s \xrightarrow{\star} s') \Rightarrow (s' \in A)$.
  - Definition of the set of reachable states of P using M:
    $\textsc{reachable}(\mathrm{M}) = Init_{\mathrm{S}} \cup \longrightarrow [\textsc{reachable}(\mathrm{M})]$
  - Main property of $\textsc{reachable}(\mathrm{M})$: $\textsc{reachable}(\mathrm{M}) \subseteq A$
  - Characterization of $\textsc{reachable}(\mathrm{M})$:
    $\textsc{reachable}(\mathrm{M}) = \mathrm{FP}(\textsc{reachable}(\mathrm{M}))$

Problems

- Proving automatically $\textsc{reachable}(\mathrm{M}) \subseteq A$:

Problems

- Proving automatically $\textsc{reachable}(\mathrm{M}) \subseteq A$: undecidable . . . no program is able to prove it automatically!

Problems

- Proving automatically $\textsc{reachable}(\mathrm{M}) \subseteq A$: undecidable . . . no program is able to prove it automatically!
- Proving automatically $\textsc{reachable}(\mathrm{M}) \subseteq A$:

Problems

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

Problems

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

Problems

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

Problems

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

Problems

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

Problems

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

Decidability or Undecidability

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 2: $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^\star$.

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 2: $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^\star$.
- 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

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 2: $w \in \mathcal{L}(G)$ where G is a grammar over the finite set of alphabet symbols Σ and $\mathcal{L}(G) \subseteq \Sigma^\star$.
- 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 functions over natural numbers: $\mathcal{F} = \mathbb{N} \nrightarrow \mathbb{N}$.
  - $\mathcal{C} \subseteq \mathcal{F}$: the set of computable (or programmable) functions
  - $f \in \mathcal{C} = \{\Phi_0, \Phi_1, \ldots, \Phi_n, \ldots\}$: the set of computable functions is countable.
  - The problem $x \in dom(\Phi_y)$ is not decidable and it expresses the correctness of programs.

Some observations

Implicit versus explicit

- Writing $101 = 5$ may have a meaning

Some observations

Implicit versus explicit

- Writing *101* $=$ *5* may have a meaning
- On the left of the symbol $=$, *101* is the code of the number n (in base 2); on the right, the code of n is its representation in base 10.
- $n_{10} = 5$ and $n_2 = 101$
- Verification: $base(2, 10, 101) = 1.2^2 + 0.2^1 + 1.2^0 = 5_{10}$

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 ?

Intuition

- A program P *produces* results from data, according to a semantics:
  - STATES is the set of all the states of P: $STATES = \mathrm{x} \to \mathbb{Z}$, where x denotes the variables of P.
  - s_0 and s_f are two states of STATES: $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from a state s_0 and produces a state s_f.
  - For a current state s of P, we write $s(\mathrm{x}) = x$ to distinguish the variable x from its current value in s:

Intuition

- A program P *produces* results from data, according to a semantics:
  - STATES is the set of all the states of P: $STATES = \mathsf{x} \to \mathbb{Z}$, where x denotes the variables of P.
  - s_0 and s_f are two states of STATES: $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from a state s_0 and produces a state s_f.
  - For a current state s of P, we write $s(\mathsf{x}) = x$ to distinguish the variable x from its current value in s:

$$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$

Intuition

- A program P *produces* results from data, according to a semantics:
  - STATES is the set of all the states of P: $STATES = \mathsf{x} \rightarrow \mathbb{Z}$, where x denotes the variables of P.
  - s_0 and s_f are two states of STATES: $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from a state s_0 and produces a state s_f.
  - For a current state s of P, we write $s(\mathsf{x}) = x$ to distinguish the variable x from its current value in s:

$$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$

  - $\mathcal{D}(P)(s_0) = s_f$ defines the following relation over the set of values:

$$x_0 \xrightarrow{\mathsf{P}} x_f$$

Intuition

- A program P *produces* results from data, according to a semantics:
  - STATES is the set of all the states of P: $STATES = \mathsf{x} \to \mathbb{Z}$, where x denotes the variables of P.
  - s_0 and s_f are two states of STATES: $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from a state s_0 and produces a state s_f.
  - For a current state s of P, we write $s(\mathsf{x}) = x$ to distinguish the variable x from its current value in s:
    $$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$
  - $\mathcal{D}(P)(s_0) = s_f$ defines the following relation over the set of values:
    $$x_0 \xrightarrow{\mathsf{P}} x_f$$
- A program P *satisfies* a contract (pre,post):
  - P transforms a variable x from an initial value x_0 and produces a final value x_f: $x_0 \xrightarrow{\mathsf{P}} x_f$
  - x_0 satisfies pre: $\mathsf{pre}(x_0)$
  - x_f satisfies post: $\mathsf{post}(x_0, x_f)$
  - $\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Verification of the contract

A program P *satisfies* a contract (pre,post):

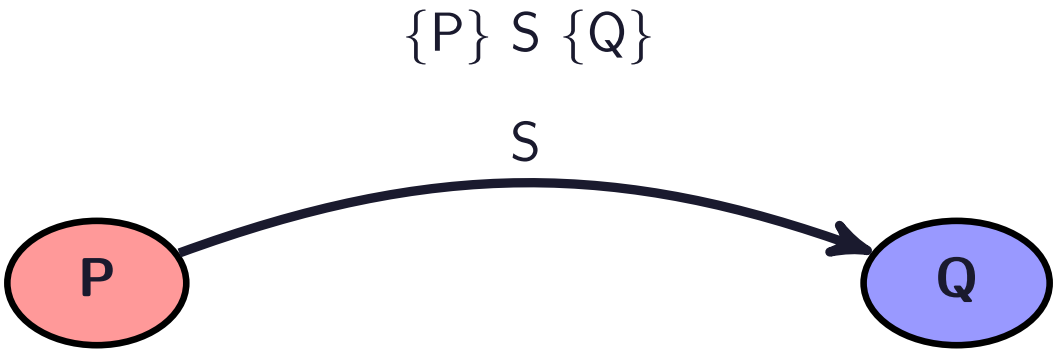
- P transforms a variable x from an initial value x_0 and produces a final value x_f: $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre: $\mathsf{pre}(x_0)$, and x_f satisfies post: $\mathsf{post}(x_0, x_f)$
- $\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

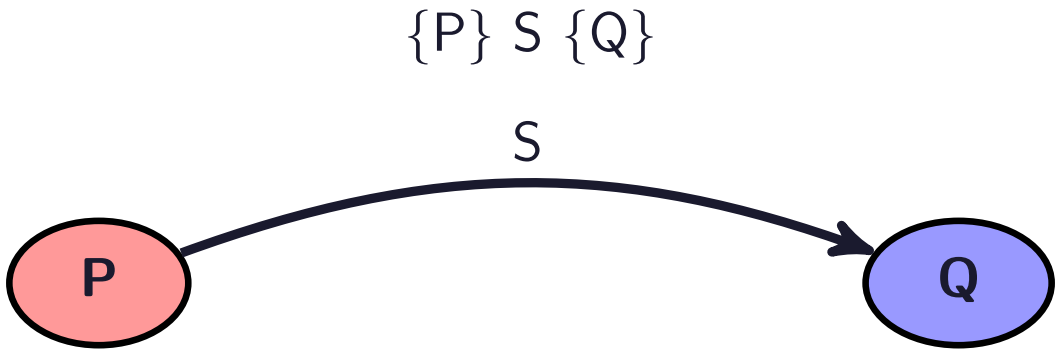
begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

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

Asserted Program {P} S {Q}

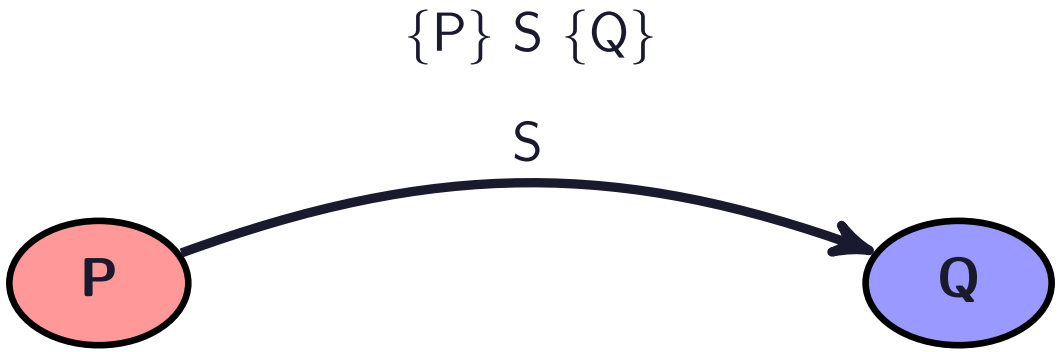


Asserted Program {P} S {Q}



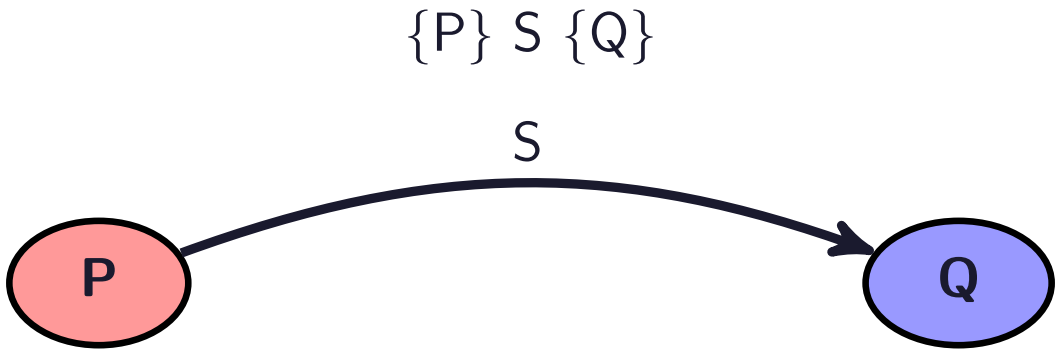
- {P} S {Q}: *asserted program*

Asserted Program {P} S {Q}



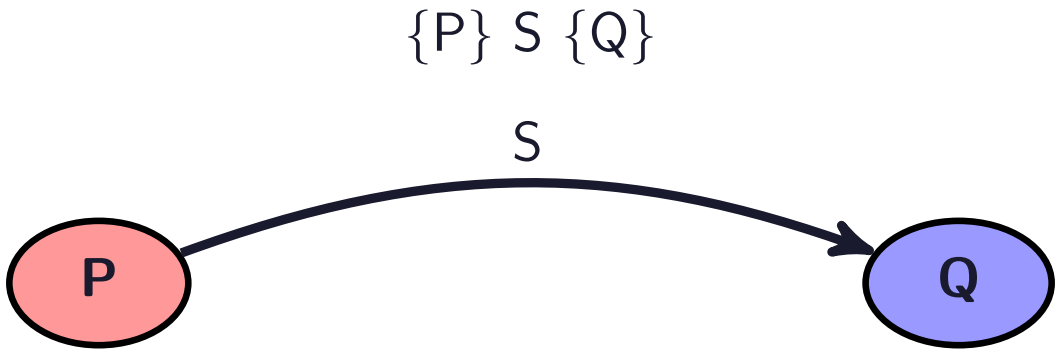
- {P} S {Q}: *asserted program*
- P $\Rightarrow WP(S)$(Q): *logical formula*

Asserted Program {P} S {Q}



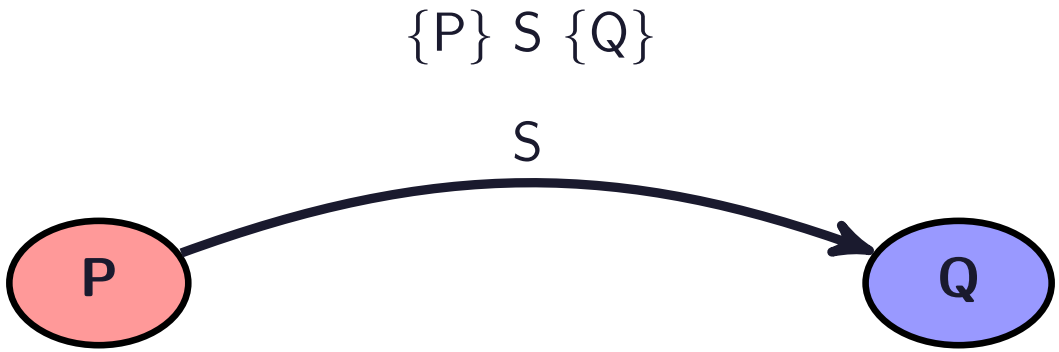
- {P} S {Q}: *asserted program*
- P $\Rightarrow WP(S)$(Q): *logical formula*
- C(S) $\vdash$ P $\Rightarrow WP(S)$(Q): *logical formula*

Asserted Program {P} S {Q}



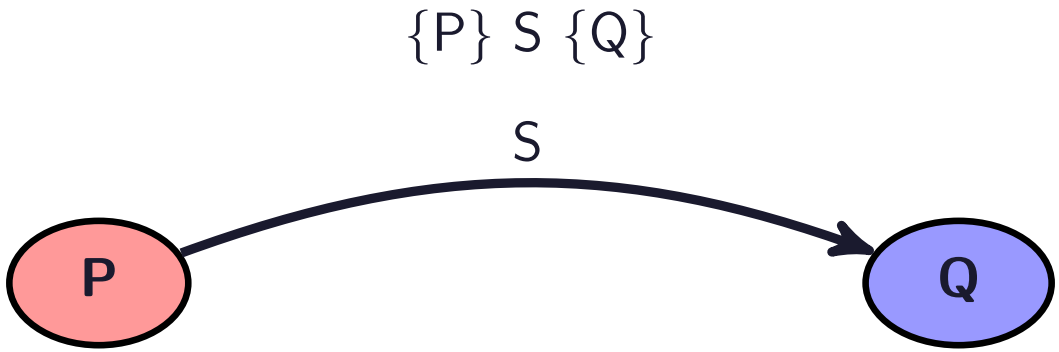
- {P} S {Q}: *asserted program*
- P $\Rightarrow WP(S)$(Q): *logical formula*
- C(S) $\vdash$ P $\Rightarrow WP(S)$(Q): *logical formula*
- $SP(S)$(P) $\Rightarrow$ Q: *logical formula*

Asserted Program {P} S {Q}



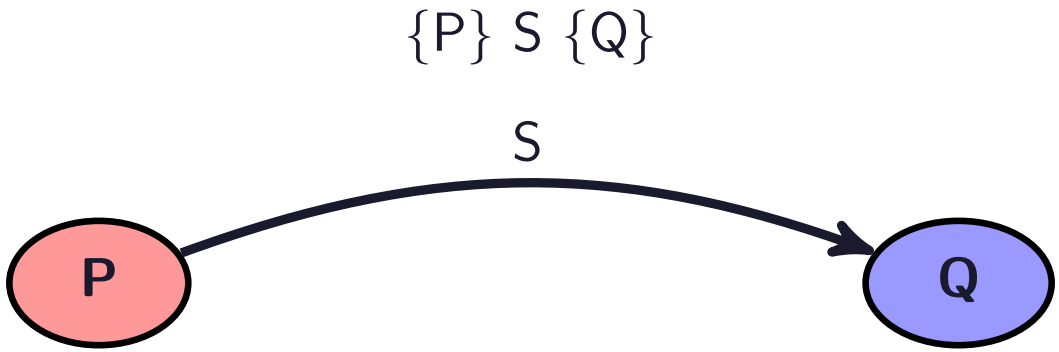
- {P} S {Q}: *asserted program*
- P $\Rightarrow WP(S)$(Q): *logical formula*
- C(S) $\vdash$ P $\Rightarrow WP(S)$(Q): *logical formula*
- $SP(S)$(P) $\Rightarrow$ Q: *logical formula*
- C(S) $\vdash SP(S)$(P) $\Rightarrow$ Q: *logical formula*

Asserted Program {P} S {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*

Asserted Program {P} S {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*

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.

Part 4

Topics of the course

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser
- Tools

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser
- Tools
  - ☺ TLA/TLA$^+$ with TLC,

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser
- Tools
  - ☺ TLA/TLA$^+$ with TLC, Frama-C

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser
- Tools
  - ☺ TLA/TLA$^+$ with TLC, Frama-C

Outline of lectures, tutorials and lab sessions

- Organisation of the unit: 10 lectures of 2 hours
- Contents:
  - ☺ Principles of modelling computer systems: transition systems
  - ☺ Properties of a computer system: safety, liveness, availability, security, dependability
  - ☺ Modelling properties of systems
  - ☺ Analysis of programs
  - ☺ Verification of system properties with the TLC model checker
  - ☺ Verification of system properties with the Rodin proof tool
  - ☺ Verification of system properties with the Frama-C analyser
- Tools
  - ☺ TLA/TLA$^+$ with TLC, Frama-C
- Assessment: a written exam and a lab session
  - ☺ Written exam and lab