Formal Methods

Cinzia Bernardeschi

Department of Information Engineering
University of Pisa, Italy

FMSS, 2021-2022

Overview

- Formal methods definition
- Specification languages
- Verification techniques
  - abstract interpretation
  - model checking
  - theorem proving
- Case studies:
  - Data leakage
  - Security protocols
  - Cyber-physical systems attacks

Formal methods definition

Formal methods are mathematically-based techniques that can be used in the design and development of computer-based systems.

Formal methods

- allow the analysis of all possible executions of the system
- improve the current techniques based on simulation and testing (mathematical proof that the system behaves as expected)
- offer the possibility for detecting vulnerability in systems or for building more secure systems by design.

Formal methods and Safety critial systems

Software used in safety critical systems is required a very high reliability

- For example, for flight-critical avionics systems in civil transport airplaines the requirement is probability of failure 10^{-9} per hour of operation. 1 failure at most every 1.000.000.000 hours of operation.
- Provide evidence of reliability. We cannot run the system for 10^{-9} hours. Long operational testing and doubts about the representativeness of the testing.

One of the best hope we have for technological support in building dependable software is the use of formal methods. (J. Knight, Fundamentals of Dependable Computing, 2012.)

Formal methods definition

A formal method consists of

- a language
  a mathematical notation or a computer language with a formal semantics
- a set of tools
  for proving properties without real executions
- a methodology for its application in industrial practice.

Formal methods definition

Formal methods are more ambitious than traditional approaches, they are more complex and the associated tools are more difficult to build.

A major problem comes from undecidability result of Computational complexity theory

- some verification problems are either impossible or very difficult to solve automatically
- For example, termination problem of programs (in the general case, there is no decision procedure that can determinate whether a program P may terminate or not)
- Another example: there is no decision procedure that can determinate whether a given instruction of program P will ever be executed

Formal methods definition

Strategies to deal with undecidability:

- expressiveness restrictions
  identify classes of systems for which verification is decidable at least in principle (example, finite systems)
- accuracy restrictions
  weaker formulation of the problem which is decidable and of interest (approximation instead of exact solutions)
- automatic restrictions
  semi-automatic verification, with human intervention at some point
  semi-decision procedures, which may terminate giving the correct result or not terminate.

Specifications

Specifications are means to describe a system, its components, its global/local environment.

- *declarative specifications*
  describe what a system should do, but not how it should do it
  it is not possible to derive automatically from these constraints an implementation of the system

- *operational specifications*
  possibly define what a system should do, and definitely define how it should do it
  it should be possible to derive automatically at least a skeleton of an implementation of the system

Formal methods for modelling and verification

- formal methods for the analysis of programs
- formal methods for the analysis of systems

Formal methods for the analysis of programs

- Semantics of the programming language
  the semantics of the language describes mathematically the meaning of the programs. *Operational semantics* as semantics of the programming language.

- Models of programs. (A model of a system is an abstraction that focus on some part of the system.)
  Control flow graphs as models of programs. The control flow graph represents the control structure of the program.

In the following:

- structured high level language (e.g., C language)
- low level languages (e.g, Java bytecode, assembly code)

A simple high level language

We consider a simple sequential language with the following syntax, where *op* stands for the usual arithmetic and logic operations

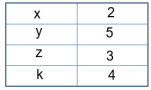
Instruction set

$$
\begin{aligned}
\mathit{exp} &::= \mathit{const} \mid \mathit{var} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \\
\mathit{com} &::= \mathit{var} := \mathit{exp} \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{com}\ \texttt{else}\ \mathit{com} \mid \\
&\quad\ \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathit{com} \mid \mathit{com}\ ;\ \mathit{com} \mid \texttt{halt}
\end{aligned}
$$

A program P is a sequence of instructions $\langle c \rangle$, where $c \in \mathit{com}$.

Standard Operational semantics

State of the program:



memory m

- $\langle c, m \rangle$
  where c is a command and m is a memory
- $\langle m \rangle$
  a single memory, in the final state.

The semantics of programs is given by means of a transition system and we call this semantics *execution semantics*.

- $Var(c)$ denote the set of variables occurring in c.
- $\mathcal{V}^{\epsilon}$ is the domain of constant values, ranged over by $k, k', \ldots,$
- for each $X \subseteq var$, the domain $\mathcal{M}^{\epsilon}_{X} = X \to \mathcal{V}^{\epsilon}$ of memories defined on X, ranged over by $m, m', \ldots$.

Transitions

The semantic rules define a relation

$$\longrightarrow^{\epsilon} \subseteq \mathcal{Q}^{\epsilon} \times \mathcal{Q}^{\epsilon}$$

where $\mathcal{Q}^{\epsilon}$ is a set of states.

A separate transition

$$\longrightarrow^{\epsilon}_{expr} \subseteq (exp \times \mathcal{M}^{\epsilon}) \times \mathcal{V}^{\epsilon}$$

is used to compute the value of the expressions.

With $m[k/x]$ we denote the memory m' which agrees with m on all variables, except on x, for which $m'(x) = k$.

Operational semantics

$$\textbf{Expr}_{const}\quad \frac{}{\langle k, m\rangle \longrightarrow^{\epsilon}_{expr} k}$$

$$\textbf{Expr}_{var}\quad \frac{}{\langle x, m\rangle \longrightarrow^{\epsilon}_{expr} m(x)}$$

$$\textbf{Expr}_{op}\quad \frac{\langle e_1, m\rangle \longrightarrow^{\epsilon}_{expr} k_1 \quad \langle e_2, m\rangle \longrightarrow^{\epsilon}_{expr} k_2 \quad k_1\ op\ k_2 = k_3}{\langle (e_1\ op\ e_2), m\rangle \longrightarrow^{\epsilon}_{expr} k_3}$$

Operational semantics

Ass $$\frac{\langle e, m\rangle \longrightarrow^{\epsilon}_{expr} k}{\langle x{:=}e, m\rangle \longrightarrow^{\epsilon} m[k/x]}$$

halt $$\frac{}{\langle \texttt{halt}, m\rangle \longrightarrow^{\epsilon} \langle m\rangle}$$

Seq$_1$ $$\frac{\langle c_1, m\rangle \longrightarrow^{\epsilon} \langle m'\rangle}{\langle c_1;c_2, m\rangle \longrightarrow^{\epsilon} \langle c_2, m'\rangle}$$

Seq$_2$ $$\frac{\langle c_1, m\rangle \longrightarrow^{\epsilon} \langle c_2, m'\rangle}{\langle c_1;c_3, m\rangle \longrightarrow^{\epsilon} \langle c_2;c_3, m'\rangle}$$

Operational semantics

$$\textbf{If}_{true} \quad \frac{\langle e, m\rangle \longrightarrow^{\epsilon}_{expr} true}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \longrightarrow^{\epsilon} \langle c_1, m\rangle}$$

$$\textbf{If}_{false} \quad \frac{\langle e, m\rangle \longrightarrow^{\epsilon}_{expr} false}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \longrightarrow^{\epsilon} \langle c_2, m\rangle}$$

$$\textbf{While}_{true} \quad \frac{\langle e, m\rangle \longrightarrow^{\epsilon}_{expr} true}{\langle \texttt{while } e \texttt{ do } c, m\rangle \longrightarrow^{\epsilon} \langle c; \texttt{while } e \texttt{ do } c, m\rangle}$$

$$\textbf{While}_{false} \quad \frac{\langle e, m\rangle \longrightarrow^{\epsilon}_{expr} false}{\langle \texttt{while } e \texttt{ do } c, m\rangle \longrightarrow^{\epsilon} \langle m\rangle}$$

Given a program $P = \langle c \rangle$ and an initial memory $m \in \mathcal{M}^{\epsilon}_{Var(c)}$, we denote by

$E(P, m)$

the transition system defined by $\longrightarrow^{\epsilon}$ starting from the initial state $\langle c, m \rangle$.

Actually, since the program is deterministic, there exists at most one final state, i.e. a state $\langle m \rangle$ for some memory m.

Transition system

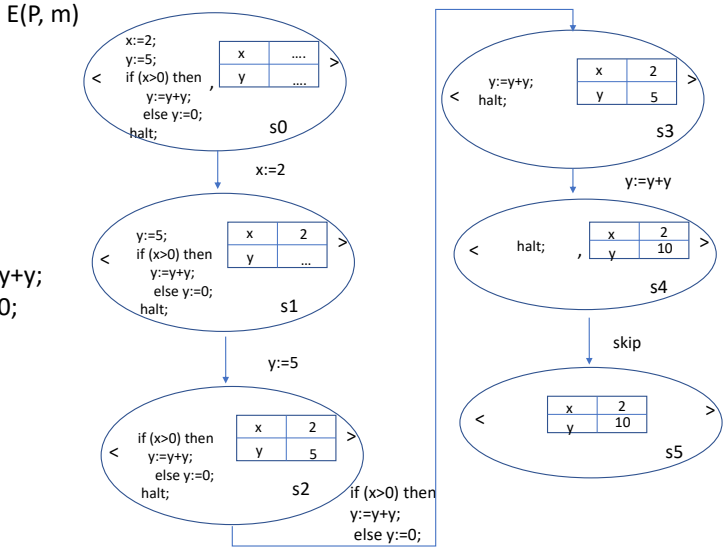
A transition system is a tuple $TS = (S, Act, \rightarrow, I)$ such that

- S is a set of states (the state space)
- Act is a set of actions
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation
  $s \rightarrow_a s'$ if the system moves from state s to s' by executing action a
- $I \subseteq S$ is the set of initial states

Often, transition systems are drawn as directed graphs with states represented by vertices and transitions represented by edges

An example

1: x:=2;
2: y:=5;
3: if (x>0)
4: then y:=y+y;
5: else y:=0;
6: halt;



Control Flow Graph

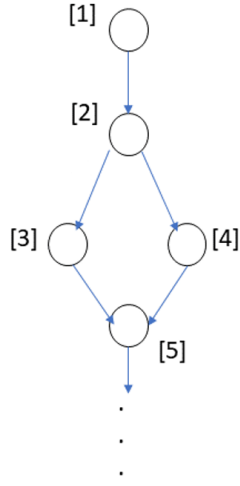
The control flow graph of a program $P =< c >$ is a directed graph $(V; E)$, where V is a set of nodes and $E : VxV$ is a set of edges connecting nodes.

Nodes correspond to instructions. Moreover, there is an initial node and a final node that represent the starting point and the final point of an execution.
E contains the edge $(i; j)$ if and only if the instruction at address j can be immediately executed after that at address i.

The control flow graph does not contain information on the semantics of the instructions.

An example

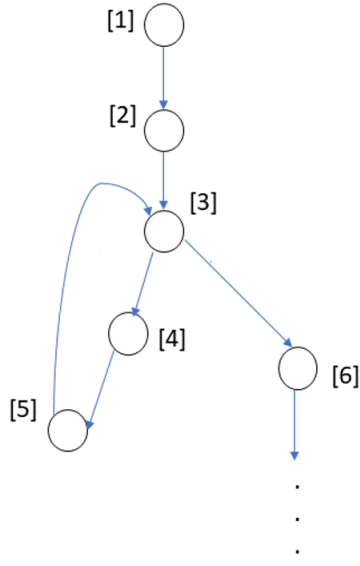
```
1 :  y:=5;
2 :  if y > 1
     then
3 :    x:=1;
     else
4 :    x:=0;
5 :  z:=x;
6 :  ....
```



C++ generation of the CFG with Visual Studio. Gcc developer-options:
-fdump-tree-cfg -blocks -vops

An example

1 : y:=5;
2 : x:=2;
3 : `while` $x > 0$
4 : y:=$y + y$;
5 : x:=$x - 1$;
6 : z:=y;
7 : .....



Java language

```
3  public class Hello {
4    public static void main(String[] args) {
5    }
6
7    public void stampa(String s){
8           System.out.println("Outside");
9
10     try { System.out.println("Inside try");}
11     catch (ArithmeticException e) {System.out.println("Print catch");}
12
13    System.out.println("Print ...");
14    }
15 }
```

Java bytecode

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: return

  public void stampa(java.lang.String);
    Code:
       0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc       #3 // String Outside
       5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: getstatic     #2 // Field java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #5 // String Inside try
      13: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      16: goto          28
      19: astore_2
      20: getstatic     #2 // Field java/lang/System.out:Ljava/io/PrintStream;
      23: ldc           #7 // String Print catch
      25: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      28: getstatic     #2 // Field java/lang/System.out:Ljava/io/PrintStream;
      31: ldc           #8 // String Print ...
      33: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      36: return

```

Java bytecode

A subset of the instruction set

Bytecode instructions operate on the operand stack.
Variables in memory are named registers.

`pop`	Pop top operand stack element.
`dup`	Duplicate top operand stack element.
op	Pop two operands off the operand stack, perform the operation *op* $\in$ { `add, mult, compare ..` }, and push the result onto the stack.
`const` *d*	Push constant *d* onto the operand stack.

Java bytecode

`load` *x*	Push the value of the register *x* onto the operand stack.
`store` *x*	Pop a value off the operand stack and store it into local register *x*.
`if`*cond j*	Pop a value off the operand stack, and evaluate it against the condition *cond* = { `eq, ge, null, ...` }; branch to *j* if the value satisfies *cond*.
`goto` *j*	Jump to *j*.

Standard Operational semantics

The domain of the states of the standard semantics is ($\mathcal{A}$ denotes the set of addresses):

$$\mathcal{Q}^{\epsilon} = \mathcal{A}^{\epsilon} \times \mathcal{M}^{\epsilon} \times \mathcal{S}^{\epsilon}.$$

A bytecode B consists of a sequence of instructions (assume 1 is the address of first instruction), and is executed starting from a memory and an empty operand stack.

A state is given by the value of three variables, *PC*, *MEM* and *STACK*, where
- *PC* is the program counter,
- *MEM* is the memory, and
- *STACK* is the operand stack (λ is the empty stack).

We denote by $\langle i, m, s \rangle$ the state labeled by $PC = i, MEM = m, STACK = s$.

Standard Operational semantics rules

$$\textbf{op} \quad \frac{c[i] = \texttt{op}}{\langle i, m, k_1 \cdot k_2 \cdot s\rangle \longrightarrow^{\epsilon} \langle i+1, m, (k_1\ op\ k_2) \cdot s\rangle}$$

$$\textbf{pop} \quad \frac{c[i] = \texttt{pop}}{\langle i, m, k \cdot s\rangle \longrightarrow^{\epsilon} \langle i+1, m, s\rangle}$$

$$\textbf{push} \quad \frac{c[i] = \texttt{push k}}{\langle i, m, s\rangle \longrightarrow^{\epsilon} \langle i+1, m, k \cdot s\rangle}$$

$$\textbf{load} \quad \frac{c[i] = \texttt{load x}}{\langle i, m, s\rangle \longrightarrow^{\epsilon} \langle i+1, m, m(x) \cdot s\rangle}$$

Standard Operational semantics rules

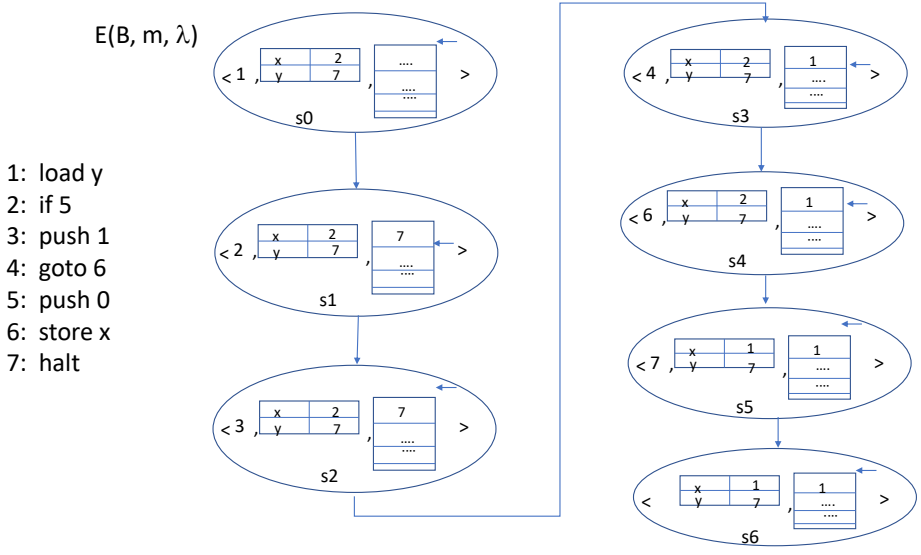
$$\textbf{store} \quad \frac{c[i] = \texttt{store x}}{\langle i, m, k \cdot s\rangle \longrightarrow^{\epsilon} \langle i+1, m[k/x], s\rangle}$$

$$\textbf{if}_{\textit{false}} \quad \frac{c[i] = \texttt{if j}}{\langle i, m, 0 \cdot s\rangle \longrightarrow^{\epsilon} \langle i+1, m, s\rangle}$$

$$\textbf{if}_{\textit{true}} \quad \frac{c[i] = \texttt{if j}}{\langle i, m, k \neq 0 \cdot s\rangle \longrightarrow^{\epsilon} \langle j, m, s\rangle}$$

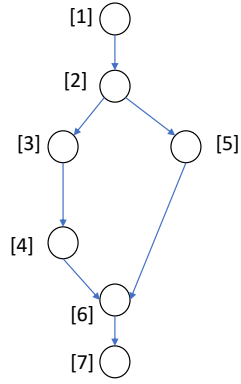
$$\textbf{goto} \quad \frac{c[i] = \texttt{goto j}}{\langle i, m, s\rangle \longrightarrow^{\epsilon} \langle j, m, s\rangle}$$

Transition system



A bytecode and its CFG

```
1: load y
2: if 5
3: push 1
4: goto 6
5: push 0
6: store x
7: halt
```



The CFG contains the edge (i, j) if and only if the instruction at address j can be immediately executed after that at address i. ([6] is the point at which the branches starting at the conditional instruction [2] join)

Java bytecode Instructions

`pop`	Pop top operand stack element.
`dup`	Duplicate top operand stack element.
α*op*	Pop two operands with type α off the operand stack, perform the operation *op* $\in$ { `add, mult, compare ..` }, and push the result onto the stack.
α`const` *d*	Push constant *d* with type α onto the operand stack.
α`load` *x*	Push the value with type α of the register *x* onto the operand stack.
α`store` *x*	Pop a value with type α off the operand stack and store it into local register *x*.
`if`*cond j*	Pop a value off the operand stack, and evaluate it against the condition *cond* = { `eq, ge, null, ...` }; branch to *j* if the value satisfies *cond*.
`goto` *j*	Jump to *j*.

Java bytecode Instructions

`getfield` *C.f*	Pop a reference to an object of class *C* off the operand stack; fetch the object's field *f* and put it onto the operand stack.
`putfield` *C.f*	Pop a value *k* and a reference to an object of class *C* from the operand stack; set field *f* of the object to *k*.
`invoke` *C.mt*	Pop value *k* and a reference *r* to an object of class *C* from the operand stack; invoke method *C.mt* of the referenced object with actual parameter *k*.
α`return`	Pop the α value off the operand stack and return it from the method.

The bytecode of a method is a sequence B of instructions.

When a method is invoked (`invoke` instruction), it executes with a new empty stack and with an initial memory where all registers are undefined except for the first one, register $x0$, that contains the reference to the object instance on which the method is called, and register $x1$, that contains the actual parameter.

When the method returns, control is transferred to the calling method: the caller's execution environment (operand stack and local registers) is restored and the returned value, if any, is pushed onto the operand stack.

An example

The bytecode corresponds to a method *mt* of a class `A`. Suppose that register *x*1 (the parameter of *A.mt*) contains a reference to an object of another class `B`. Note that register *x*0 contains a reference to `A`. After the bytecode has been executed, the final value of field `f1` of the object of class `A` is 0 or 1 depending on the value of field `f2` of the object of class `B`.

```
0 : aload    x0
1 : aload    x1
2 : getfield B.f2
3 : ifge     6
4 : iconst   0
5 : goto     7
6 : iconst   1
7 : putfield A.f1
8 : iconst   1
9 : return
```

Program graphs

Program graphs

A program graph is a control flow graph with edges labelled by actions - two special nodes are identified: the initial, which represents the starting of the execution, and the final node, which represents a point where the execution will have terminate

Program graphs are supposed to compute some output values based on some input.

A program graph consists of

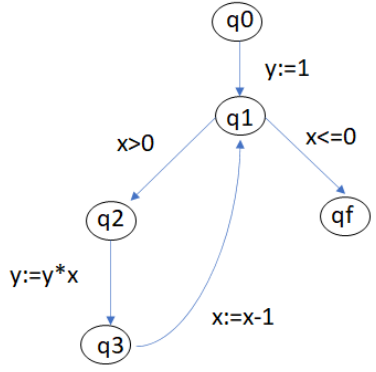
- Q: a finite set of nodes
- $q_0, q_f \in Q$: the initial and the final node, respectively
- Act: a set of action
- $\rightarrow \subseteq Q \times Act \times Q$ a finite set of edges.

F. Nielson, H.R. Nielson, Formal Methods. Springer, 2019.

Program graph for the factorial function

Input: x Output: y Function: $y = x!$

$Q = \{q_0, q_1, q_2, q_3, q_f\}$
$Act = \{y := 1, x > 0, x <= 0, x := x - 1, y := y * x\}$
$\rightarrow = \{q_0 \rightarrow_{y:=1} q_1, \cdots\}$



Program graphs

The language for the program graph above is:
$y := 1 \;\; (x > 0 \;\; y := y * x \;\; x := x - 1)^* \;\; x <= 0$

To deal with different programming languages, the language *Guarded Commands* introduced by Dijkstra in 1975 is used, and the techniques used can be extended to the more familiar languages.

Program graphs

The semantics of a program graph consists of

- a memory $m \in M$
- a semantic function specifying the meaning of the actions
  given a memory before executing the action, returns the memory after the action.

Configuration.
A configuration is a pair (q, m), with $q \in Q$ a and $m \in M$. Memory m is assigned to state q of the graph.

The memory assigned to q_0, is named the initial memory. The factorial is computed on the value assigned to x in the initial memory.

Properties

Deterministic system

- a program graph and its semantics constitute a *deterministic* system whenever for each complete execution sequence, starting from the same initial memory, the value computed is the same.

Sufficient condition

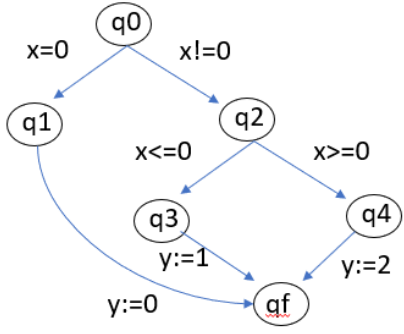
- a program graph and its semantics constitute a deterministic system whenever: distinct edges with the same source node have satisfaction conditions for the enabling of the action that do not overlap

The program graph and its semantics for the factorial function is deterministic.

Properties

The previous condition is not necessary for obtaining a deterministic system.

$x >= 0$ and $x <= 0$ overlap when x is equal to 0.



In state $q2$, the value of x is always different from 0.

Properties

Evolving system

- A program does not stop in a stuck configuration (evolving system). Weaker condition that *the program always compute a value (because the program may loop)*.

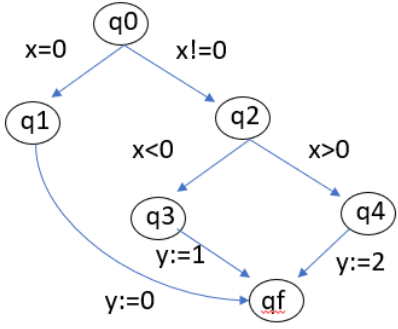
Sufficient condition

- a program graph and its semantics constitute an evolving system if for every non-final node and every memory there is an edge leaving it such that the satisfaction condition for the enabling of the action is satisfied.

Properties

The previous condition is not necessary for obtaining an evolving system.

In state $q2$, $x > 0$ and $x < 0$ have a gap exactly when x is equal to 0.

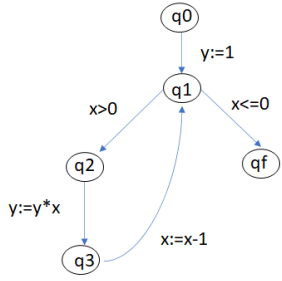


In state $q2$, the value of x is always different from 0.

Program verification

Verify the correctness of a program

- attach a predicate to each node of the program graph
- the predicate should be satisfied whenever the node is reached during execution
- high complexity, expressive power of predicates;
- cannot be fully automated



Introduction of var that cannot be changed
by the actions of the program (underlined).
Partial predicate assignment P (condition on coverability of the program graph: $q0, qf \in dom(P)$, each loop contains a node in $dom(P)$):
@$q0$: $x = \underline{n} \wedge \underline{n} \geq 0$
@$q1$: $\underline{n} \geq x \wedge x \geq 0 \wedge y * fact(x) = fact(\underline{n})$
@qf: $y = fact(\underline{n})$

Program analysis

- A fully automated approach to prove properties.

- Express approximate behaviours of programs.
- It is a static technique, because information are obtained without the executing the program on real values.

Basic rule:

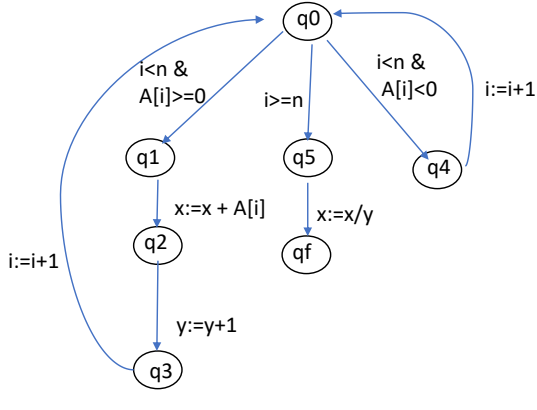
we use abstractions of the memories rather than the real memories.
Generally, abstractions are built by considering properties of the values.

For example:
natural numbers abstracted to the property of being odd or even.
integer numbers abstracted to their sign: positive, zero or negative.

An example

Computing the average of non-negative elements of an array.

Input: A[3]
Output: y = average of non-negative elements of A



initial memory m

x	0
y	0
i	0
n	3
A[0]	0
A[1]	4
A[2]	7

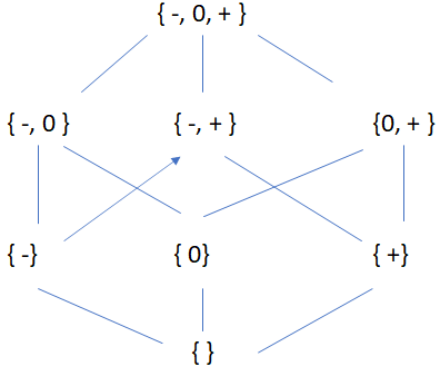
initial configuration (q0, m)

Property we want to analyse

- Sign of the sum of elements of an array

Abstract memories: values are replaced by their sign, assuming $\mathit{Sign} = \{-, 0, +\}$.

Power of *Sign*:



Examples of abstract memories

x	0
y	0
i	0
n	3
A[0]	0
A[1]	4
A[2]	7

Each element of the array A
Is abstracted into its sign.

abstract memory

x	0
y	0
i	0
n	+
A[0]	0
A[1]	+
A[2]	+

Another solution:
We can abstract the elements
of the array A into a single element
which is the union of the signs.

abstract memory

x	0
y	0
i	0
n	+
A	{0,+}

Abstract executions

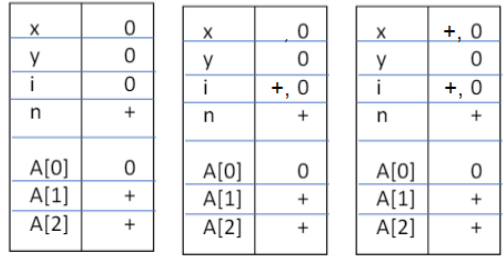
Example: Addition of individual signs

$+^\circ$	-	0	+
-	{-}	{-}	{-, 0, +}
0	{-}	{0}	{+}
+	{-, 0, +}	{+}	{+}

Program analysis

memories associated to the node "while"

$i := 0;$
$x := 0;$
$while(i < 3) \quad \rightarrow$
$x := x + A[i];$
$i = i + 1;$



x	0
y	0
i	0
n	+
A[0]	0
A[1]	+
A[2]	+

x	0
y	0
i	+, 0
n	+
A[0]	0
A[1]	+
A[2]	+

x	+, 0
y	0
i	+, 0
n	+
A[0]	0
A[1]	+
A[2]	+

Program analysis

The analysis of the program is an assignment of abstract memories with each node in the program graph

- The collection of abstract memories associated with a node should be an over-approximation of the set of memories that could occur at that node in an execution sequence.
- If a concrete memory m occurs at node q at some point during an execution sequence then the collection of abstract memory at q must contain the corresponding abstract memory, but may contain additional abstract memories as well.

Program analysis

The analysis of the program is an assignment A of abstract memories with each node in the program graph

$$A : Q \rightarrow PowerSet(Mem)$$

where $PowerSet(Mem)$ is the set of collections of abstract memories.