Formal Methods

Cinzia Bernardeschi

Department of Information Engineering
University of Pisa, Italy

FMSS, 2019-2020

Overview

- Relation between dependability and security
- Formal methods definition
- Specification languages
- Verification techniques
  - model checking
  - abstract interpretation
  - theorem proving
- Case studies:
  - Malware analysis
  - Data leakage
  - 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 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 of the system
- a methodology for its application in industrial practice.

Formal methods for modelling and verification

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

Formal methods for the analysis of programs

- Semantics of the programming language
- Control flow graph of a program

The semantics of the language describes mathematically the behaviour of the program.

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

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

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

Control Flow Graph

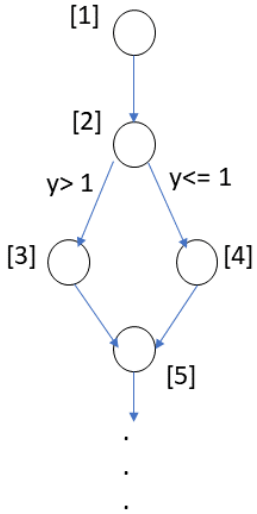
The control flow graph of a program $P =< c >$ is a directed graph $(V; E)$, where V is a set of nodes and $E : VxV$ is a set of edges connecting nodes.

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

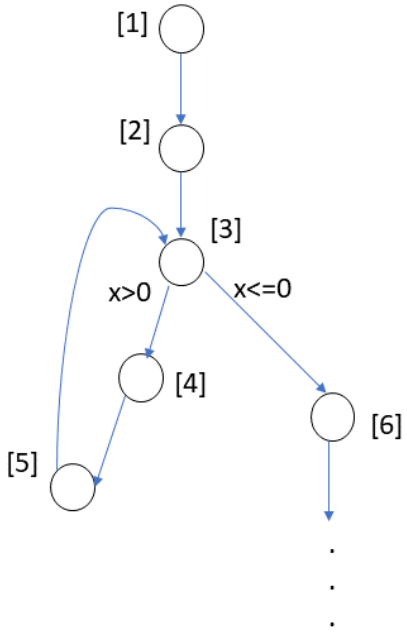
```
1 :  y:=5;
2 :  if y > 1
     then
3 :    x:=1;
     else
4 :    x:=0;
5 :  z:=x;
6 :  ....
```



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

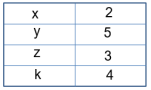
An example

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



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 $m'(x) = k$.

The symmetric rule for the conditional command (false condition) are omitted.

Operational semantics

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

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

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

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

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

Operational semantics

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

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

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

$$\textbf{Seq}_1 \quad \frac{\langle c_1, m\rangle \longrightarrow^{\epsilon} \langle m'\rangle}{\langle c_1; c_2, m\rangle \longrightarrow^{\epsilon} \langle c_2, m'\rangle} \qquad \textbf{Seq}_2 \quad \frac{\langle c_1, m\rangle \longrightarrow^{\epsilon} \langle c_2, m'\rangle}{\langle c_1; c_3, m\rangle \longrightarrow^{\epsilon} \langle c_2; c_3, m'\rangle}$$

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.

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

Instruction set

`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 Instruction set

`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.

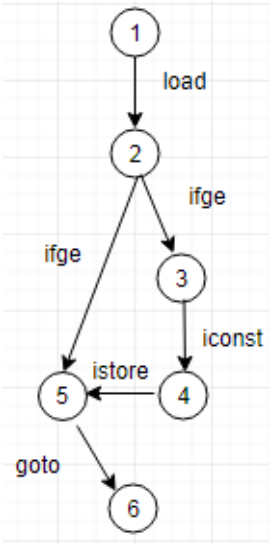
An example

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
```

A code and its CFG

```
0 :  .....
1 :  load x
2 :  ifge 5
3 :  iconst_1
4 :  istore_2
5 :  goto 6
6 :  .....
```



Standard Operational semantics

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

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

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

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

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

Standard Operational semantics

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

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

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

Standard Operational semantics

x: 5
y: 1

state: <program counter, memory, operand stack>

$$\langle PC, [MEM(x)\ MEM(y)], STACK \rangle$$

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

$$\langle 0, [5,1], \lambda \rangle$$
$$\downarrow_{\mathbf{load}}$$
$$\langle 1, [5,1], 1 \rangle$$
$$\downarrow_{\mathbf{if_{true}}}$$
$$\langle 4, [5,1], \lambda \rangle$$
$$\downarrow_{\mathbf{push}}$$
$$\langle 5, [5,1], 0 \rangle$$
$$\downarrow_{\mathbf{store}}$$
$$\langle 6, [0,1], \lambda \rangle$$