

# MODEL CHECKING







# SIMPLE PROMELA INTERPRETER MODEL CHECKER

---

- verifica formale di sistemi concorrenti e/o distribuiti (programmi concorrenti, sistemi operativi, protocolli di interazione e comunicazione etc.)
  - Bell Labs – Gerard J. Holzmann
  - 2002 - ACM Software System Award
- **PROMELA (Protocol/Process Meta Language)**
  - creazione dinamica di processi concorrenti
  - comunicazione sincrona/asincrona fra processi tramite canali
- strumenti per convertire programmi Java/C in modelli per Spin
- esistono alcune interfacce utenti
- <http://spinroot.com/spin/whatispin.html>  
<http://spinroot.com/spin/Man/index.html>

# Model Checking

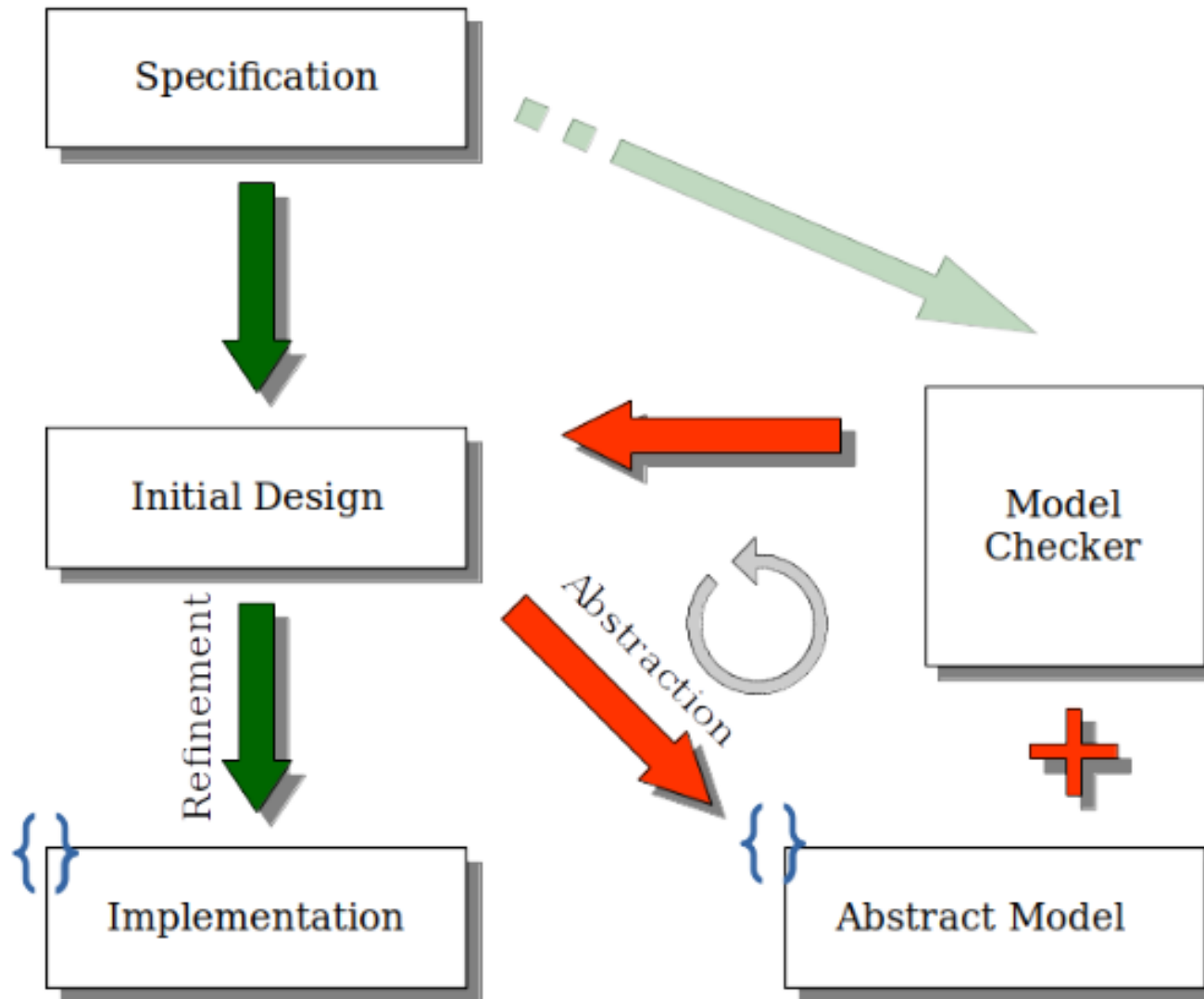
- $M \models \varphi$

verifica formale ed automatica del fatto che il modello  $M$  di un sistema rispetti specifiche proprietà formali espresse mediante una collezione di formule logiche  $\varphi$

- difetti riscontrabili:

- **starvation**: processi che non accedono mai a risorse
- **deadlock**: insieme di processi bloccati in attesa circolare
- **race conditions**: violazioni di accessi mutuamente esclusivi
- **constraints violation**: buffer overrun, overflows, ...
- **under-specified model**: comportamento inattesi
- **over-specified model**: codice ineseguito / stati irraggiungibili
- ...

# “Classic” Model Checking





## es. spin\_1: random **simulation**

```
active proctype P() {  
    int value = 123; // provare con byte qui ..  
    int reversed;    // .. e qui  
    reversed =  
        (value % 10) * 100 +  
        ((value / 10) % 10) * 10 +  
        (value / 100);  
    printf("value = %d, reversed = %d\n", value, reversed)  
}
```

```
$ spin nomefile.pml
```



# es. spin\_1: random **simulation**

\$ **spin** -uN file.pml      profondità limitata a N

\$ **spin** -p file.pml      mostra istruzioni eseguite dai processi

\$ **spin** -g file.pml      mostra variabili globali

\$ **spin** -l file.pml      mostra variabili locali

\$ **spin** -s file.pml      mostra istruzioni `send` eseguite su un canale

\$ **spin** -r file.pml      mostra istruzioni `receive` eseguite su un canale

# data types

Type	Values	Size (bits)
bit, bool	0, 1, false, true	1
byte	0..255	8
short	-32768..32767	16
int	$-2^{31}..2^{31} - 1$	32
unsigned	$0..2^n - 1$	$\leq 32$

- variabili inizializzate a 0 per default, ma inizializzarle al valore giusto serve per diminuire gli stati esplorati
- no **char**, no **string** no **float**

# operatori ed espressioni 1/2

Precedence	Operator	Associativity	Name
14	()	left	parentheses
14	[ ]	left	array indexing
14	.	left	field selection
13	!	right	logical negation
13	~	right	bitwise complementation
13	++, --	right	increment, decrement
12	*, /, %	left	multiplication, division, modulo
11	+, -	left	addition, subtraction
10	<<, >>	left	left and right bitwise shift

- **le espressioni in PROMELA non hanno side-effects**



# operatori ed espressioni 2/2

Precedence	Operator	Associativity	Name
9	<, <=, >, >=	left	arithmetic relational operators
8	==, !=	left	equality, inequality
7	&	left	bitwise and
6	^	left	bitwise exclusive or
5		left	bitwise inclusive or
4	&&	left	logical and
3		left	logical or
2	( -> : )	right	conditional expression
1	=	right	assignment

- variabili locali ad un processo sono visibili in tutto il processo e sono quindi implicitamente dichiarate all'inizio
- gli assegnamenti non sono espressioni
- `x++`
- ~~`a = x++`~~

# strutture di controllo

- **guarded commands** (Dijkstra)

**::** *command*

particolarmente adatto per rappresentare indeterminismo

1) **sequenza** **;**

2) **selezione** **if .. fi**

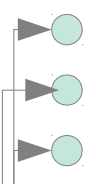
3) **ripetizione** **do .. od**

4) **goto**

5) **unless**

# sequenza ;

- ; separatore (no terminatore)

-  `x = y + 2 ;`  
`z = x * y ;`  
`printf("x=%d, z=%d\\n", x, z)`

control points

## es. spin\_2: selezione **if** .. **fi**

```
active proctype P() {  
    int a = 1, b = -4, c = 4;  
    int disc;  
    disc = b * b - 4 * a * c;  
    if  
    :: disc < 0 ;  
        printf("disc = %d: no radici reali\n", disc)  
    :: disc == 0 ;  
        printf("disc = %d: radici reali doppie\n", disc)  
    :: disc > 0 ;  
        printf("disc = %d: due radici reali\n", disc)  
    fi  
}
```



## es. spin\_3: selezione

```
active proctype P() {  
    byte giorni;  
    byte mese = 2;  
    int anno = 2000;  
    if  
    :: mese == 1 || mese == 3 || mese == 5 || mese == 7 ||  
       mese == 8 || mese == 10 || mese == 12 ->  
        giorni = 31  
    :: mese == 4 || mese == 6 || mese == 9 || mese == 11 ->  
        giorni = 30  
    :: mese == 2 && anno % 4 == 0 &&  
       (anno % 100 != 0 || anno % 400 == 0) -> giorni = 28  
    :: else -> giorni = 29  
    fi;  
    printf("mese = %d, anno = %d, giorni = %d\n", mese, anno, giorni)  
}
```

# es. spin\_4: selezione non deterministica

```
active proctype P() {  
    int a = 5, b = 5;  
    int max;  
    int ramo;  
    if  
    :: a >= b -> max = a; ramo = 1;  
    :: b >= a -> max = b; ramo = 2;  
    fi;  
    printf("Il maggiore fra %d e %d è %d seguendo il ramo %d\n",  
        a, b, max, ramo);  
}
```

- **nondeterminismo** → random simulation
- **se tutte le alternative sono false il processo si blocca fino a che qualche guardia diventa true** (cosa che può accadere solo nei processi concorrenti)

```
...  
if  
:: branch = 1;    selettore senza guardie  
:: branch = 2;  
fi;
```

```
active proctype P() {  
    int x = 15, y = 20;  
    int a, b;  
    a = x; b = y;  
    do  
    :: a > b -> a = a - b  
    :: b > a -> b = b - a  
    :: a == b -> break  
    od;  
    printf("il MCD di %d e %d è %d\n", x, y, a);  
}
```

## es. spin\_6: ripetizione con contatore

```
active proctype P() {
    int N = 11;
    int somma;
    byte i = 1;
    do
        :: i > N -> break
        :: else ->
            somma = somma + 2;
            i++;
    od;
    printf("La somma dei primi %d numeri è %d\n", N, somma);
}
```

```
active proctype P() {
    int somma = 0;
    int N = 14;
    byte i;
    for (i : 1..N) {
        somma = somma + 2;
    }
    printf("La somma dei primi %d numeri è %d\n", N, somma);
}
```



# goto



do

::  $i > N \rightarrow$  goto qui

:: else  $\rightarrow$

sum = sum + 2;

i++

od;

qui;

printf(...);

...

qui:

do

::  $i > N \rightarrow$  goto qui

:: else  $\rightarrow$  skip

sum = sum + 2;

i++

od;

# asserzioni

- lo spazio degli stati viene esplorato alla ricerca di un'eventuale violazione di una qualche specifica di correttezza
- specifiche di correttezza esprimibili come **asserzioni**  
**assert** (*espressione*)
- se *espressione* viene valutata `true` l'esplorazione prosegue, altrimenti viene generato un messaggio d'errore
- **precondizioni**: asserzioni che devono essere vere nello stato iniziale
- **postcondizioni**: asserzioni che devono essere vere in **qualunque** stato finale
- **invariante**: asserzione nel corpo di un ciclo
- `$ spin -1 filename.pml`

## es. spin\_7: asserzioni

```
active proctype P() {  
    int dividendo = 15;                                // modificare in 16  
    int divisore  = 4;  
    int quoziente, resto;  
    assert (dividendo >= 0 && divisore > 0);             // PRECONDIZIONE  
    quoziente = 0;  
    resto = dividendo;  
    do  
        :: resto >= divisore ->                          // cambiare >= in >  
            quoziente++;  
            resto = resto - divisore  
        :: else -> break  
    od;  
    printf("%d diviso per %d da %d col resto di %d\n",  
        dividendo, divisore, quoziente, resto);  
    assert (0 <= resto && resto < divisore);             // POSTCONDIZIONI  
    assert (dividendo == quoziente * divisore + resto);  
}
```

## es. spin\_8: invarianti

```
active proctype P() {  
    int dividendo = 15, divisore = 4;  
    int quoziente = 0, resto = 0;  
    int n = dividendo;  
    assert (dividendo >= 0 && divisore > 0);  
    do  
        :: n != 0 ->  
            assert (dividendo == quoziente * divisore + resto + n);  
            assert (0 <= resto && resto < divisore);  
            if  
                :: resto + 1 == resto -> quoziente++; resto = 0  
                :: else -> resto++  
            fi;  
            n--  
        :: else -> break  
    od;  
    printf("%d diviso per %d fa %d col resto di %d\n",  
        dividendo, divisore, quoziente, resto);  
    assert (dividendo == quoziente * divisore + resto);  
    assert (0 <= resto && resto < divisore);  
}
```



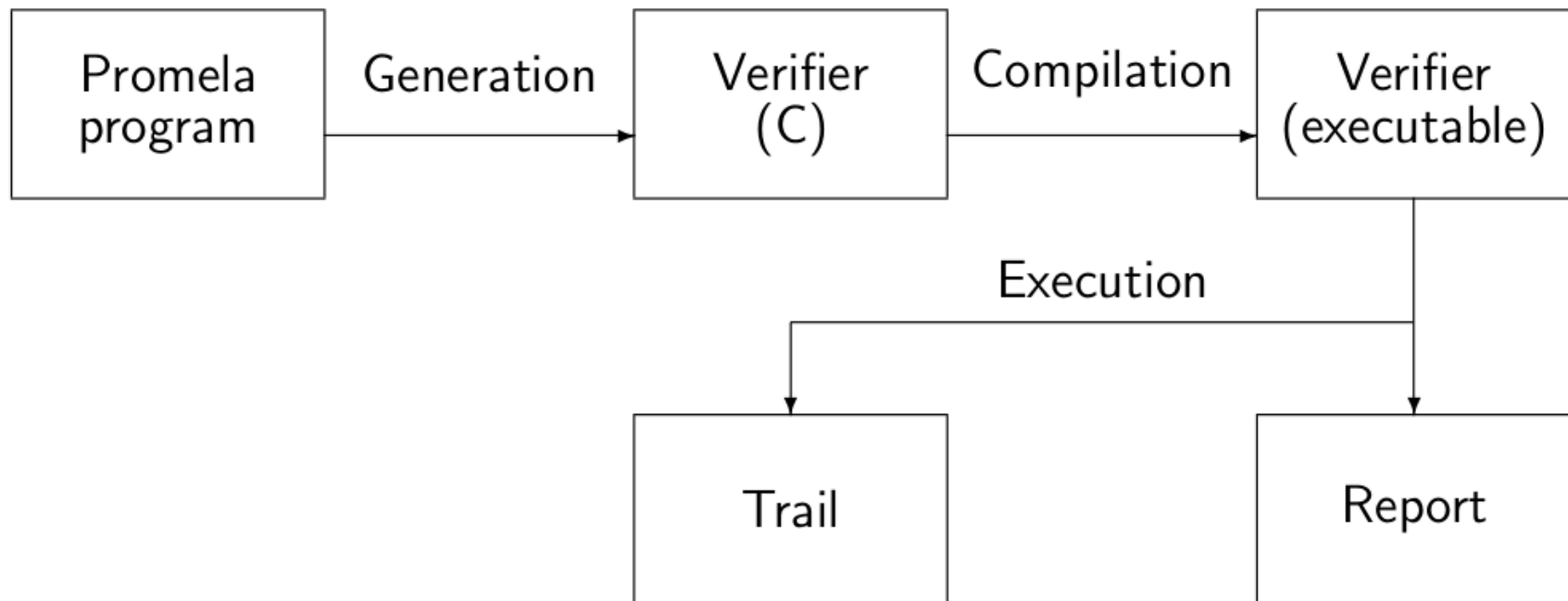


# verifica di programmi in SPIN

---

- in programmi non-deterministici (es. programmi concorrenti) SPIN deve verificare la correttezza *per tutte le possibili computazioni*, quindi deve effettuare il *backtraking* in ogni punto di scelta
- necessità di metodi efficienti per esplorare tutto lo spazio degli stati → **verificatore efficiente**

# verificatore



- 1) *generare* il **verificatore** in C a partire dal programma in Promela
- 2) *compilare* il verificatore
- 3) *eseguire* il verificatore che genera Report e Trail

```
$ spin -a filename.pml  
$ gcc -o pan pan.c // protocol analyzer  
$ ./pan // -e,tutti errori -cN, i primi N -c0, nessuno
```

## es. spin\_9: simulazione guidata

```
active proctype P() {  
    int a = 5, b = 5, max;  
    if  
    :: a >= b -> max = a;  
    :: b >= a -> max = b+1; /* ERRORE */  
    fi;  
    assert (a >= b -> max == a : max == b);  
}
```

- supportare il progettista nel processo d'individuazione dei problemi
- le informazioni utili vengono inserite nel `filename.trail`
- `filename.trail` serve per la simulazione guidata

```
$ spin -t filename.pml
```