Verifica di Sistemi Embedded

Università degli Studi di Salerno

Tecniche Automatiche per la Correttezza del Software

Professore: S. La Torre

Studente: Iodice Christian

Indice

- Introduzione ai Sistemi Embedded
- Il problema della verifica
- Verifica formale
  - Theorem Proving
  - Automated Abstraction
  - Invisible Formal Methods
  - BMC
    - CBMC (SAT-based BMC)
      - ESW-CBMC (SMT-based BMC)

Sistemi embedded

- Sistemi a microprocessore
- Special purpose
- Hardware progettato ad-hoc

Testing & debugging

- Testing e debugging hanno raggiunto i propri limiti
  - **«Program testing can be used to show the presence of bugs, but never to show their absence!»**
- Nuovi scenari per il Model Checking
  - Bounded Model Checking

Il problema della verifica

- Consuma più del 70% dell'intero budget ingegneristico
- Ottimizzata per sistemi embedded
- Deve far fonte a:
  - Complessa interazione con l'ambiente
  - Numerosi vincoli come:
    - Vincoli su risorse computazionali;
    - Vincoli sulla potenza di consumo;
    - Vincoli sull'utilizzo della memoria;
    - Vincoli real-time;
    - **Vincoli di sincronizzazione;**
  - Requisiti di alta affidabilità
    - Tempestività
    - Tolleranza ai guasti
    - Affidabilità
- La combinazione di questi fattori fanno sì che il design di sistemi embedded diventi molto complesso.

Verifica formale

- L'idea di base è costruire:
  - un **modello formale** del sistema da analizzare che abbia tutte le possibili caratteristiche del sistema
  - Un insieme di **requisiti di correttezza** che descrive le caratteristiche del sistema
- La verifica formale prova se il modello formale è conforme ai requisiti di correttezza.

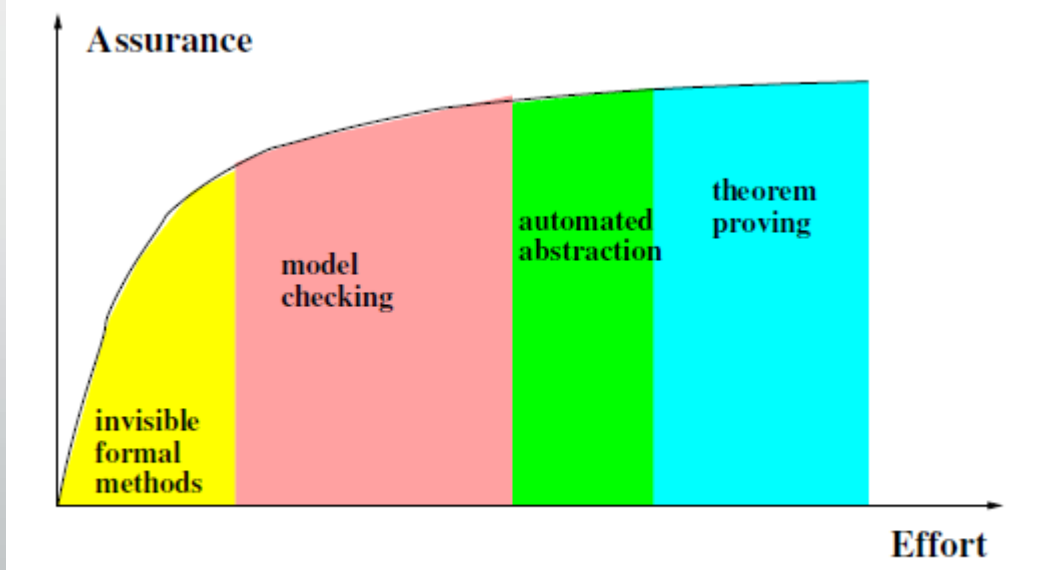


Fig. 1. A spectrum of formal methods.

Theorem Proving

- Processo iterativo e interattivo
- Si basa sulla dimostrazione di teoremi matematici
- Richiede esperienza e grande abilità per 'guidare' il tool

$$\{Q_{PRE}\}P\{R_{POST}\}$$

Q_{PRE}: *precondition*
R_{POST}: *postcondition*
P: *entire program or single function call*

- **Il theorem proving si occupa di dimostrare che una data condizione è una conseguenza logica di un insieme di istruzioni**

Theorem proving – Punti da ricordare

- Prova la correttezza del programma attraverso la derivazione di un teorema
- Esplora solo alcuni stati del sistema del modello che sono necessari per dimostrare il teorema
- L'intervento di un utente **esperto** è necessario per il processo di verifica. Per dimostrare piccoli teoremi è necessario un grande investimento umano
- E' in grado di dimostrare l'assenza di errori
- La teoria alla base di teoremi è abbastanza complessa
- Come verificare il theorem prover stesso?
- Usato nella pratica per il 'FDIV bug' nel 1994

Automated abstraction

- Sfrutta il paradigma dell'astrazione eliminando tutti i dettagli superflui
- Per i sistemi embdedded possono essere usate generiche astrazioni come data-path, abstract interpretation etc.
- Usato in pratica nel progetto SAL (Symbolic Analysis Laboratory)
  - Consiste nell'analizzare un modello astratto usando il model checking

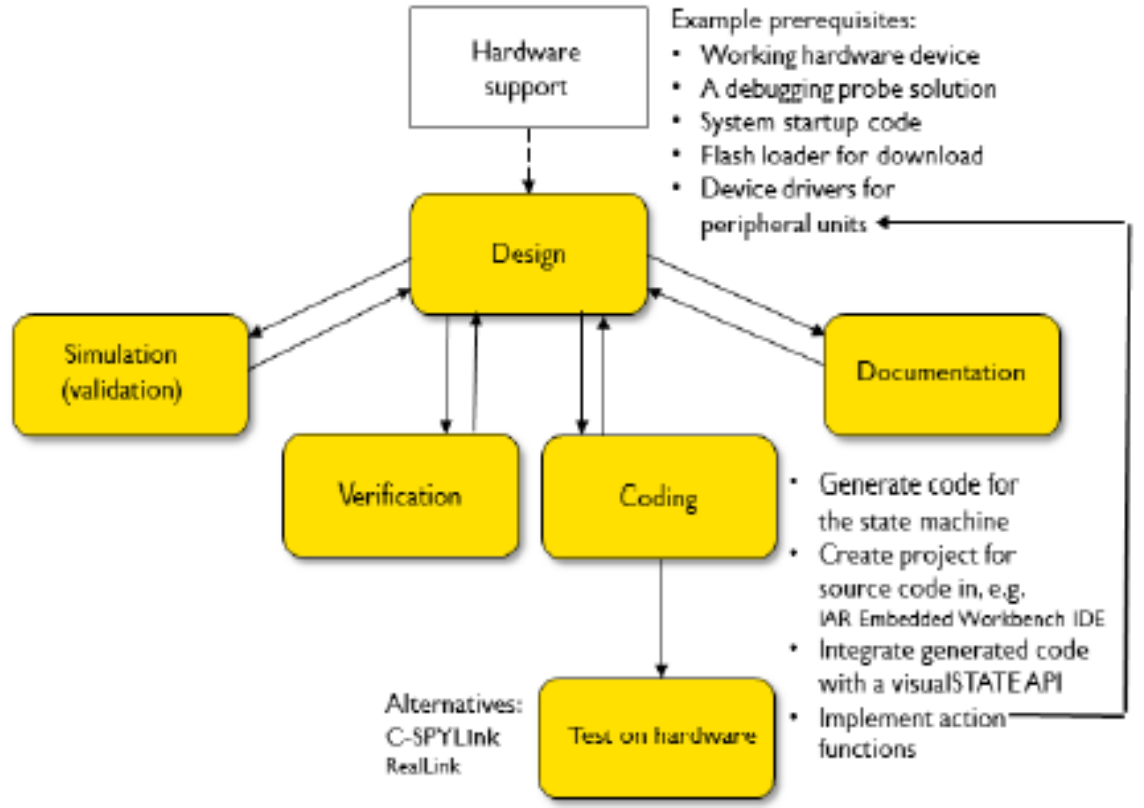
Invisible Formal Methods

- Tecnologia di verifica direttamente applicabile su modelli UML e sul software
- **Gli algoritmi e la loro matematica di base sono completamente invisibili per l'utente**.
- Riescono a trovare un limitato numero di bugs come:
  - Deadlock;
  - Transizioni in conflitto tra gli stati;
  - Stati non raggiungibili;
  - Eventi o segnali non utilizzati;
  - Transizioni non utilizzate;
  - Variabili, parametri e costanti non utilizzate;

Invisible Formal Methods – VisualSTATE

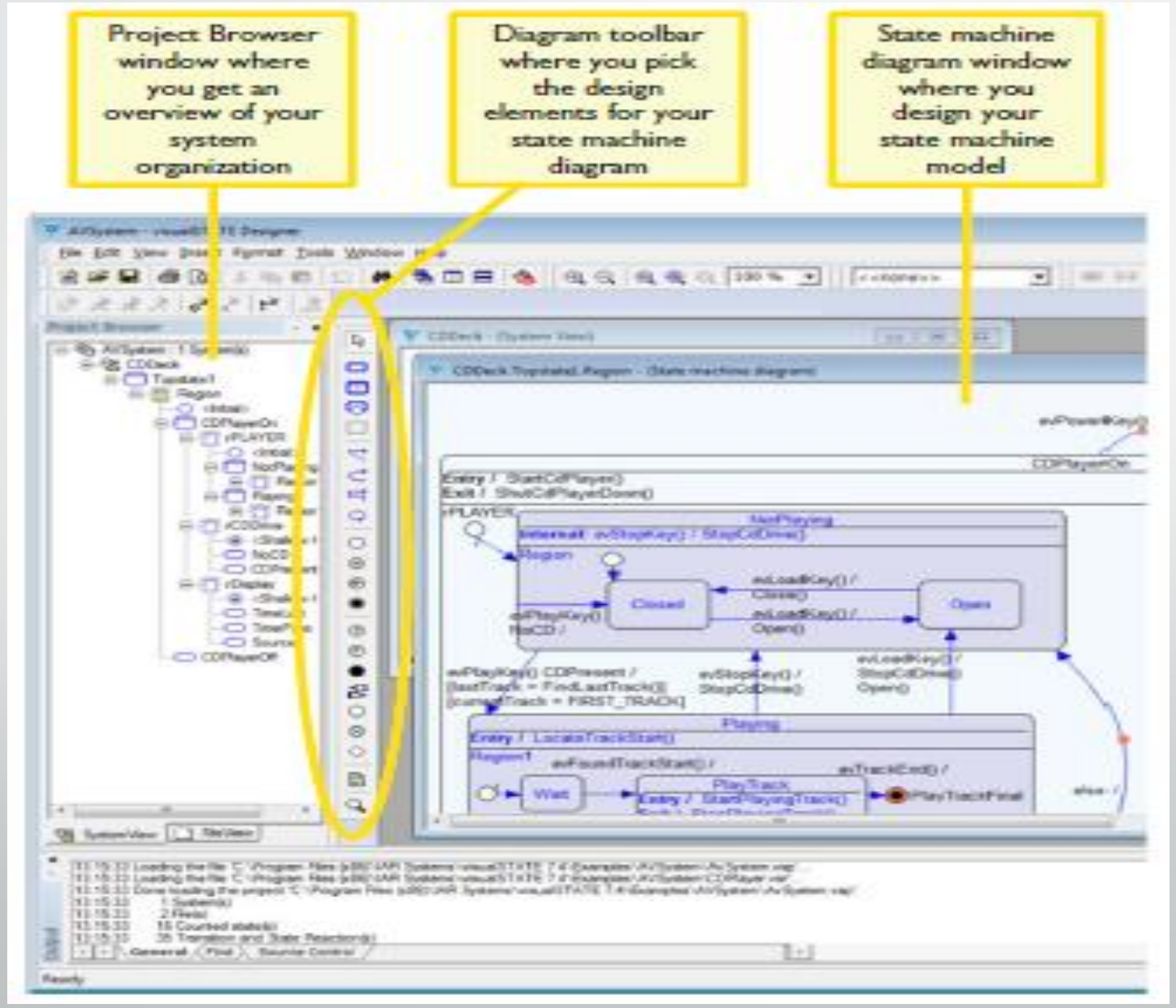
- E' una suit di tools usato per realizzare e testare applicazioni embedded
- Sfrutta un sofisticato algoritmo di verifica chiamato '**compositional backward reachability analysis**' per verificare, esaustivamente, grandi applicazioni industriali in alcuni minuti su uno standard PC
- Può essere usato su diagrammi UML standard.
- Le componenti fondamentali sono:
  - **Designer**: è un tool grafico per creare stati macchina usando la notazione UML
  - **Validator**: è un tool grafico per simulare, analizzare ed effettuare il debbugging di modelli creati con il Designer. Il validator è usato per testare le funzionalità del design
  - **Verificator**: è un tool per la verifica formale di un modello creato con il Designer
  - **Coder**: genera automaticamente codice per i modelli creati con il Designer, che può essere combinato manualmente con il codice già scritto.

VisualSTATE – Ciclo di sviluppo



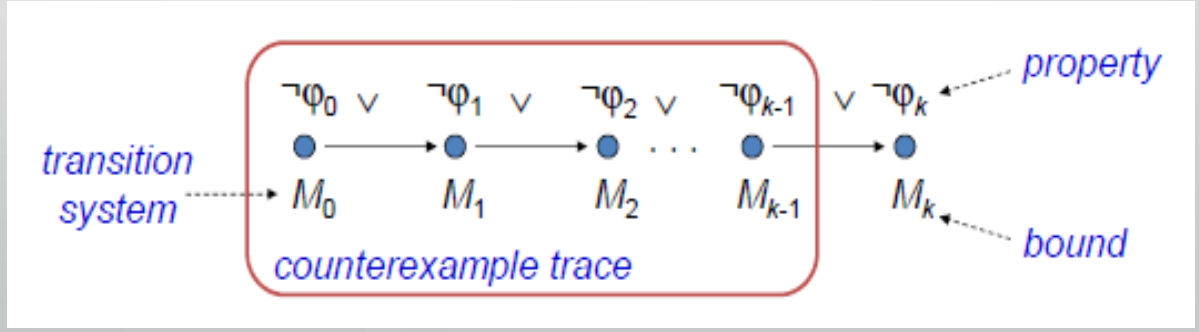
- Qui vengono sfruttati, per ogni relativa fase, i componenti descritti precedentemente

VisualSTATE – Interfaccia principale

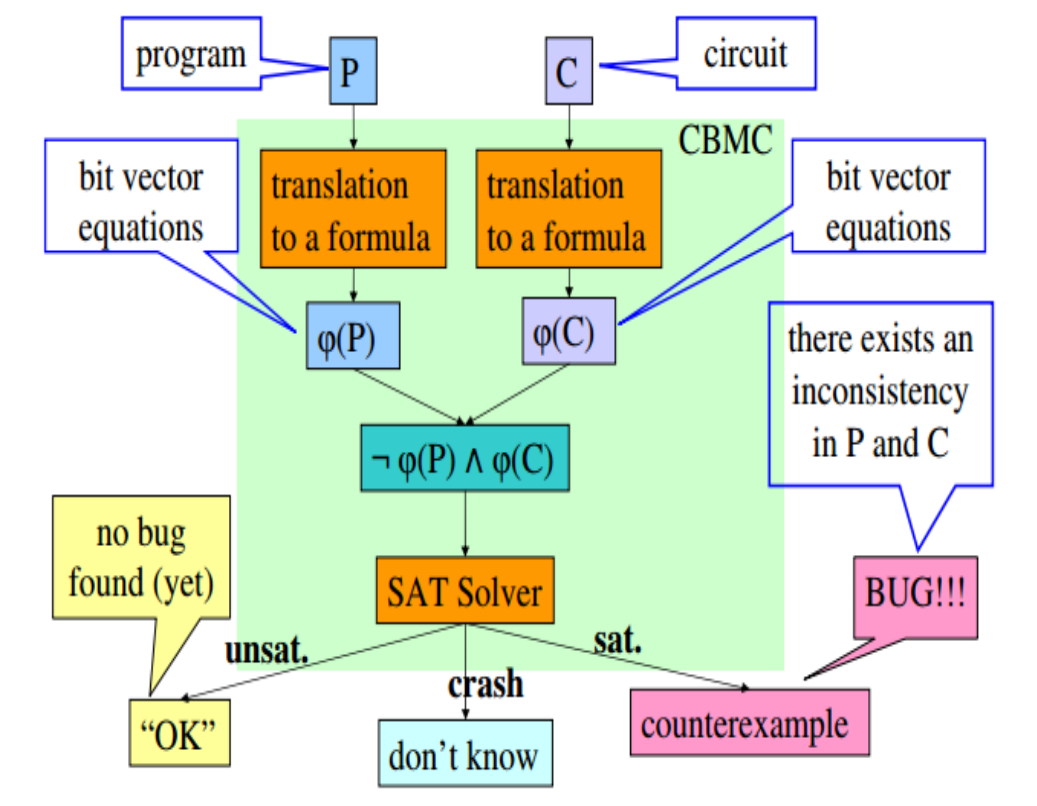


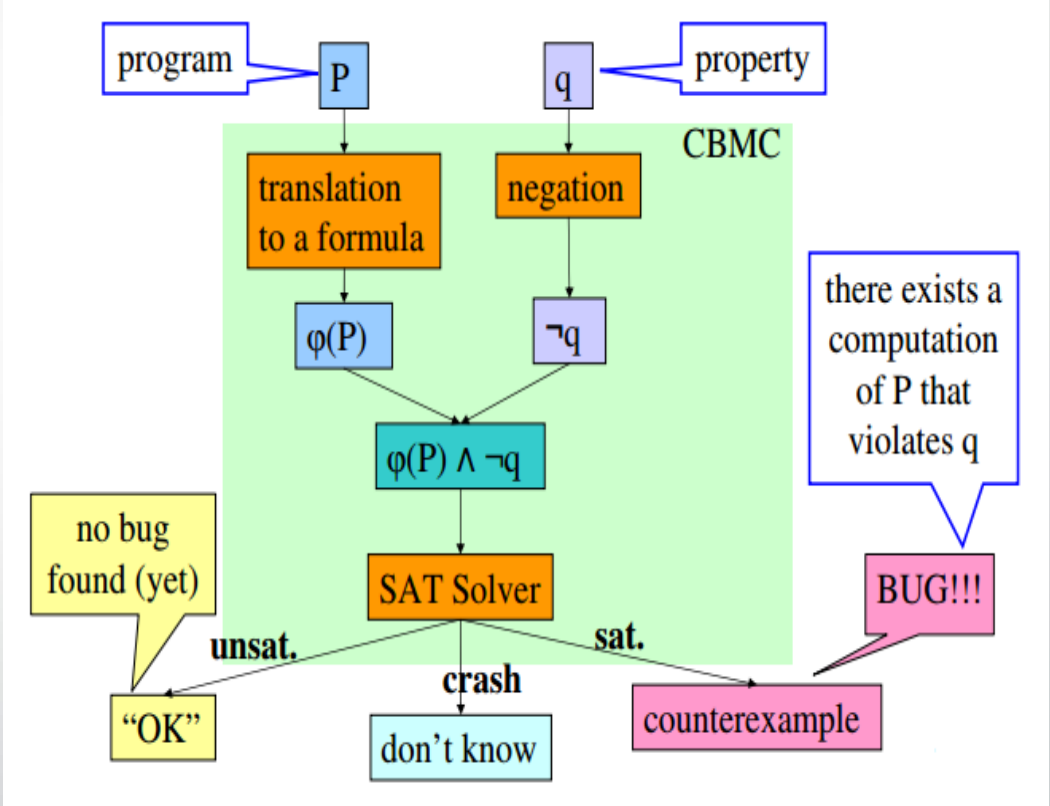
BMC- Bounded Model Checking

- Usato efficientemente per verificare software embedded
- Si basa sulla soddisfacibilità booleana (SAT)
- **Ha lo scopo di controllare la violazione di una proprietà ad una data profondità del sistema**:
  - Dati un sistema di transizioni M, una proprietà ϕ e un bound k che limita lo srotolamento, il BMC "apre" il sistema k volte, per convertirlo in una condizione di verifica ψ, espressa in una logica proposizionale, tale che ψ è soddisfacibile sse ϕ ha un controesempio di profondità minore o uguale a k.
  - Graficamente:

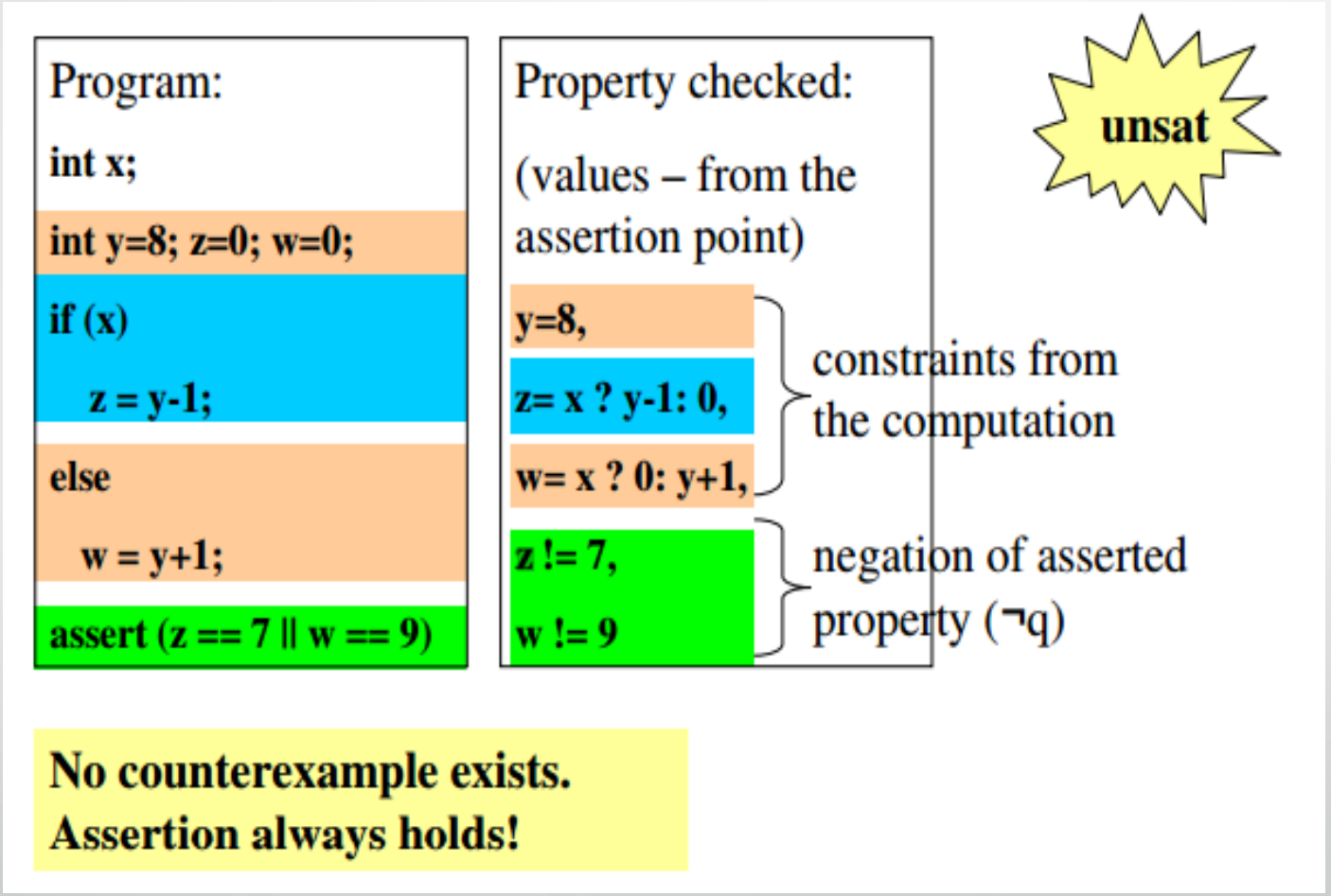


BMC & SAT-solver

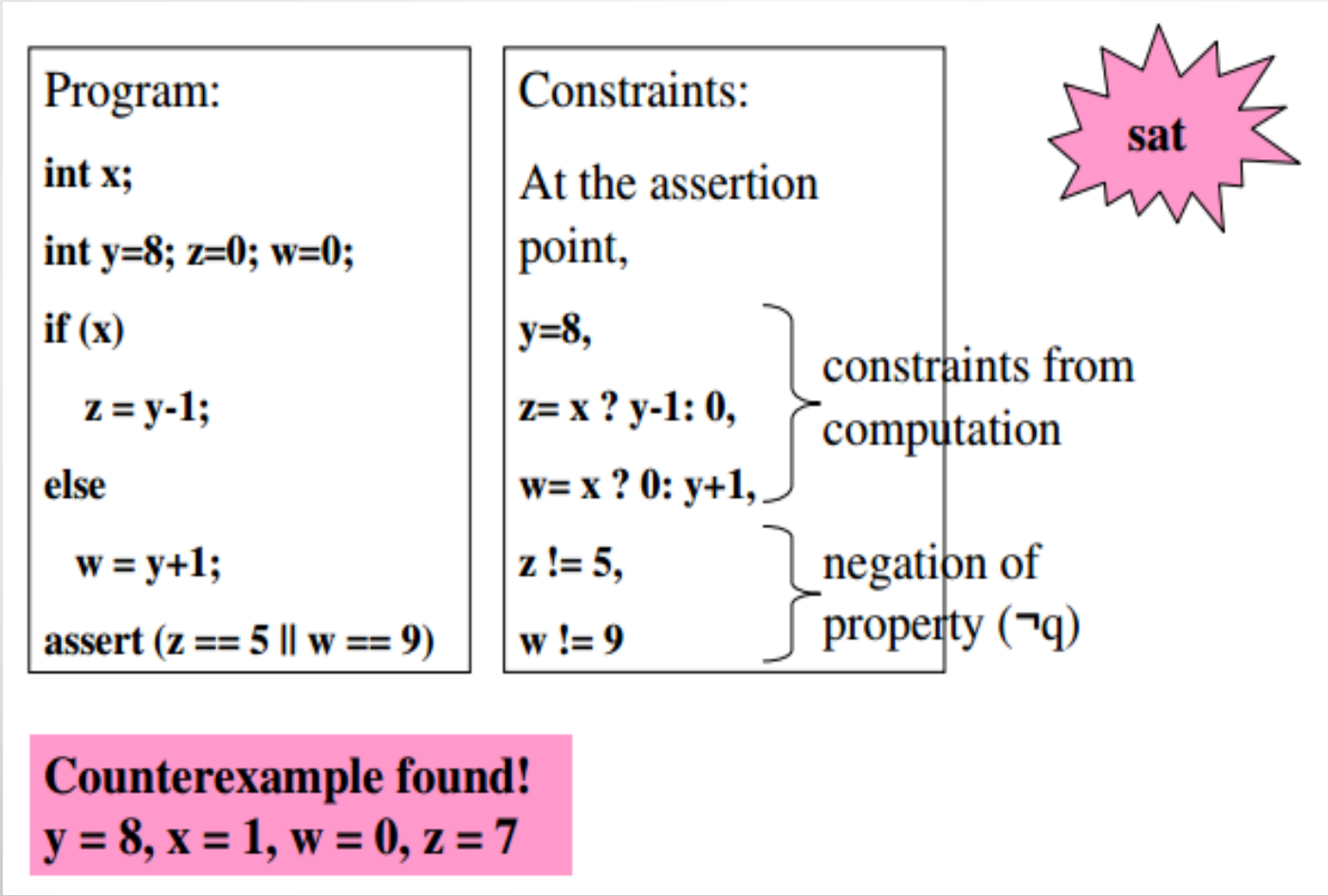




Logica funzionamento (1)



Logica funzionamento (2)

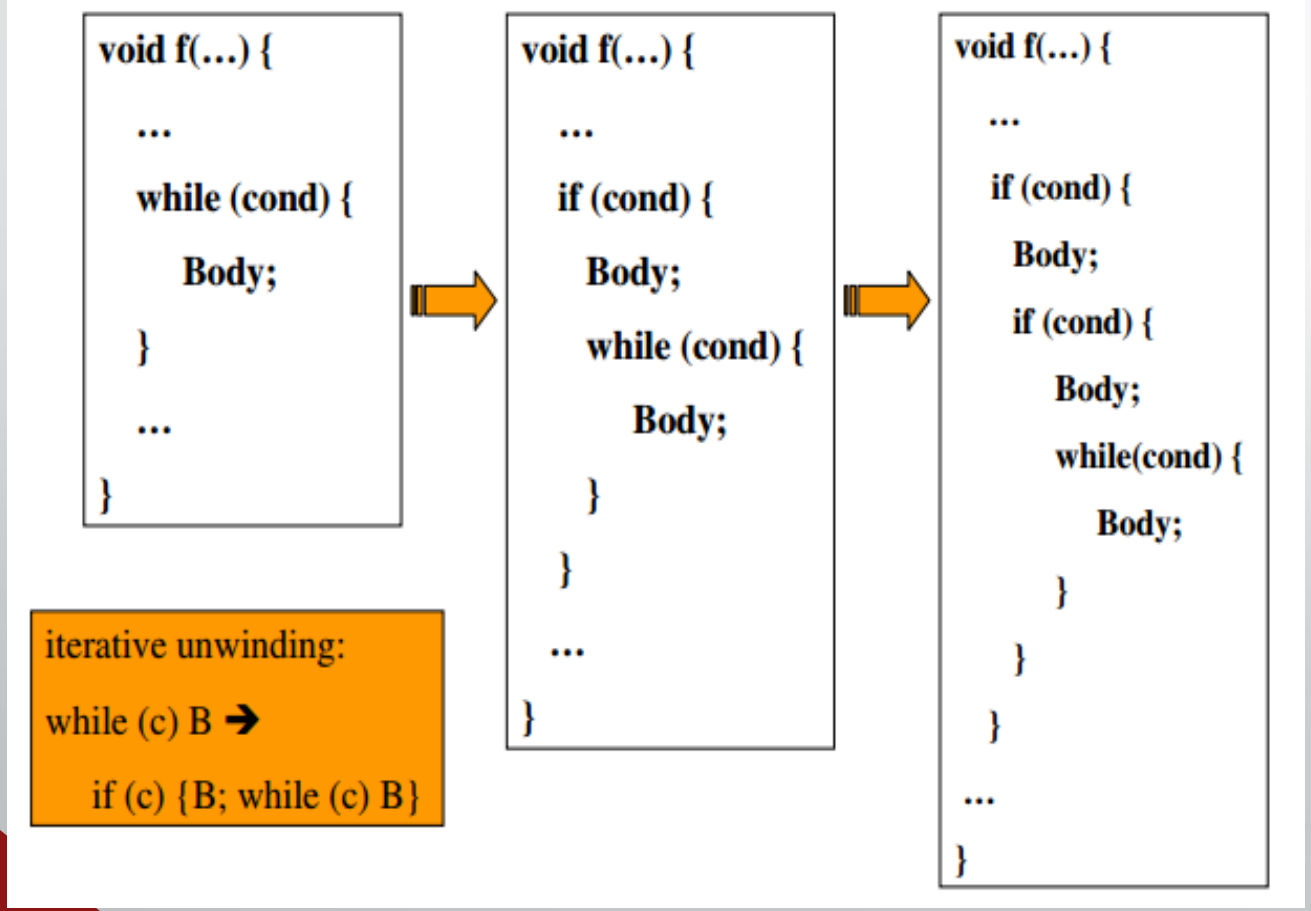


Trasformazioni per la verifica

- Prima di poter realmente verificare le proprietà bisogna realizzare le seguenti trasformazioni sul programma:
  1. Semplificare il controllo di flusso;
     A. Rimuovere i side effects (j=i++ diventerà j=i; i=i+1);
     B. Rendere il controllo del flusso esplicito (“continue” e “break” diventeranno “goto”);
     C. Semplificare i loop (“for” e “do while” diventeranno “while”).
  2. Srotolare i loop;
  3. Conversione in SSA (Single Static Assignment);
  4. Conversione in un insieme di equazioni;

2. Srotolare i loop

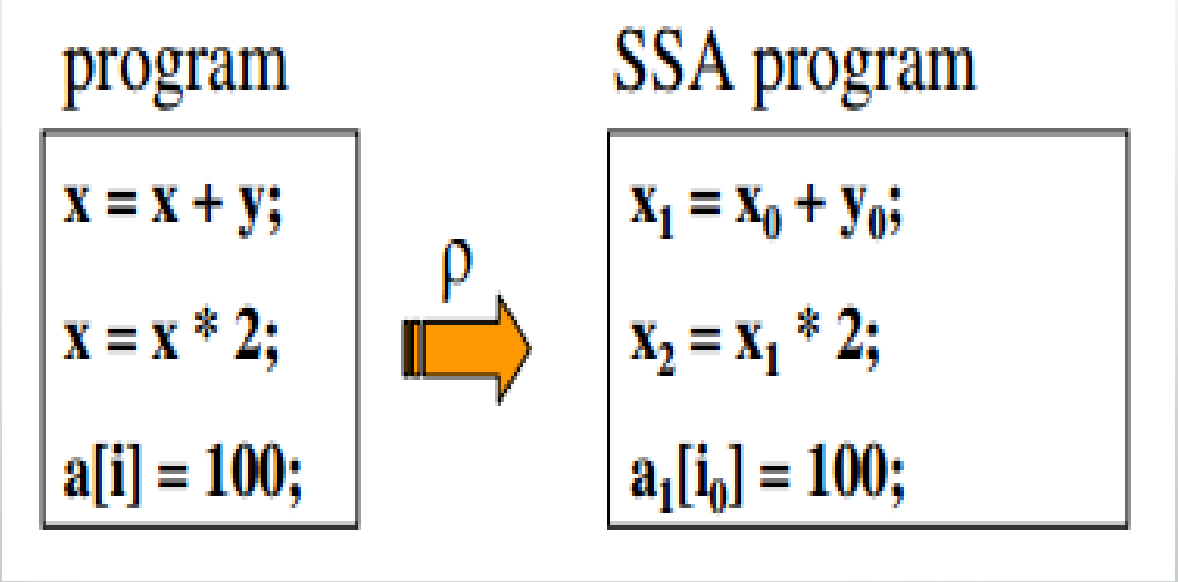
- Per fare questa operazione si ha bisogno di un bound al numero di srotolamenti.
- **Una asserzione è inserita dopo l'ultima iterazione. Essa sarà violata se il programma esegue più iterazioni che il bound permette.**




```
void f(…) {
  …
  if (cond) {
    Body;
    if (cond) {
      Body;
      if (cond) {
        Body;
        assert (!cond);
      }
    }
  }
 …
}
```

3. Conversione in SSA (1)

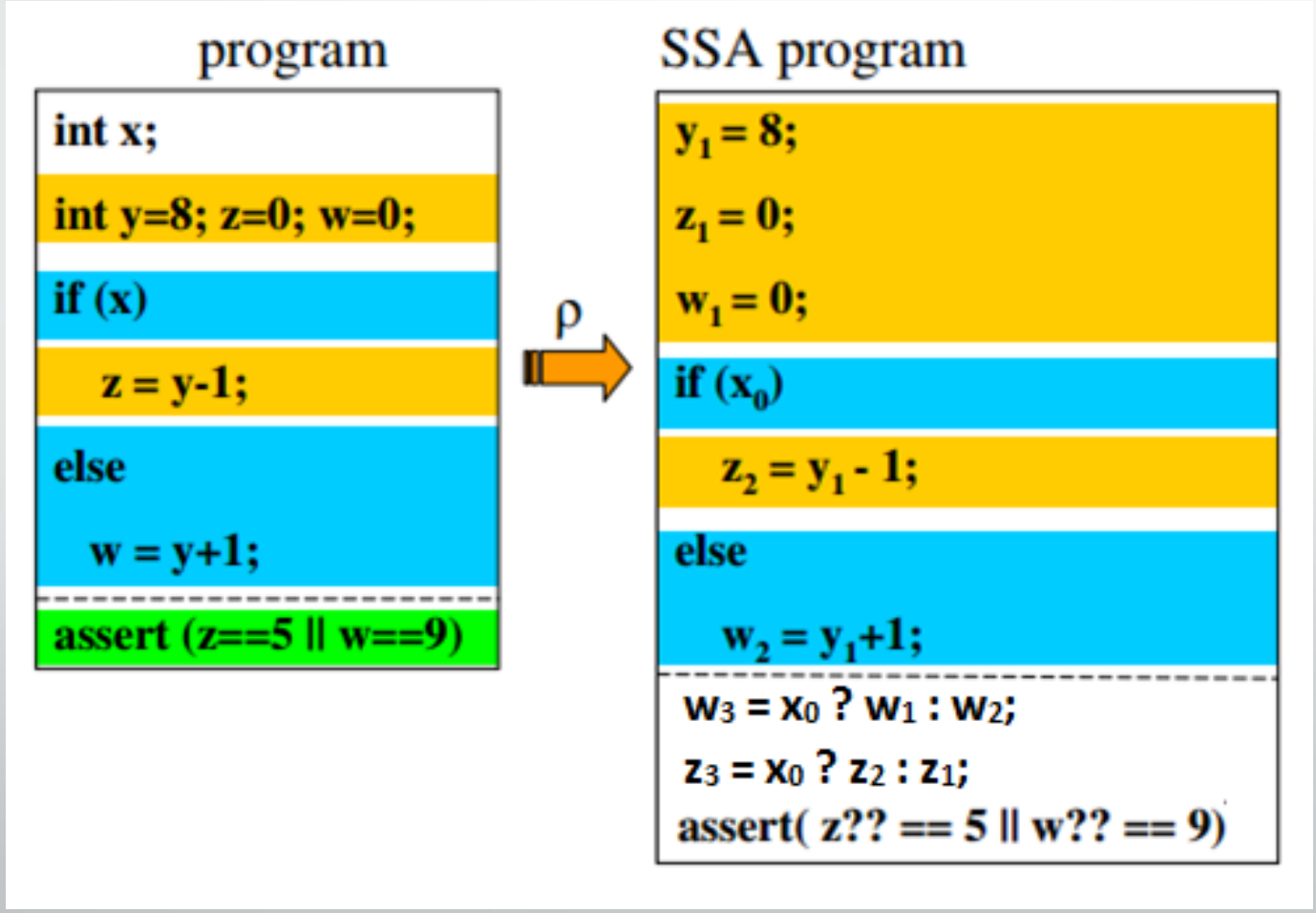
- **Idea**: convertire un programma in una formula in modo tale che ogni variabile venga assegnata solo una volta nell'intero programma;
- **Algoritmo:** Utilizzare una nuova variabile per il lato sinistro di ogni assegnazione;
  - il nome della prima variabile usata è rappresentato dal numero 0.



Problema: se il programma usa IF-ELSE?

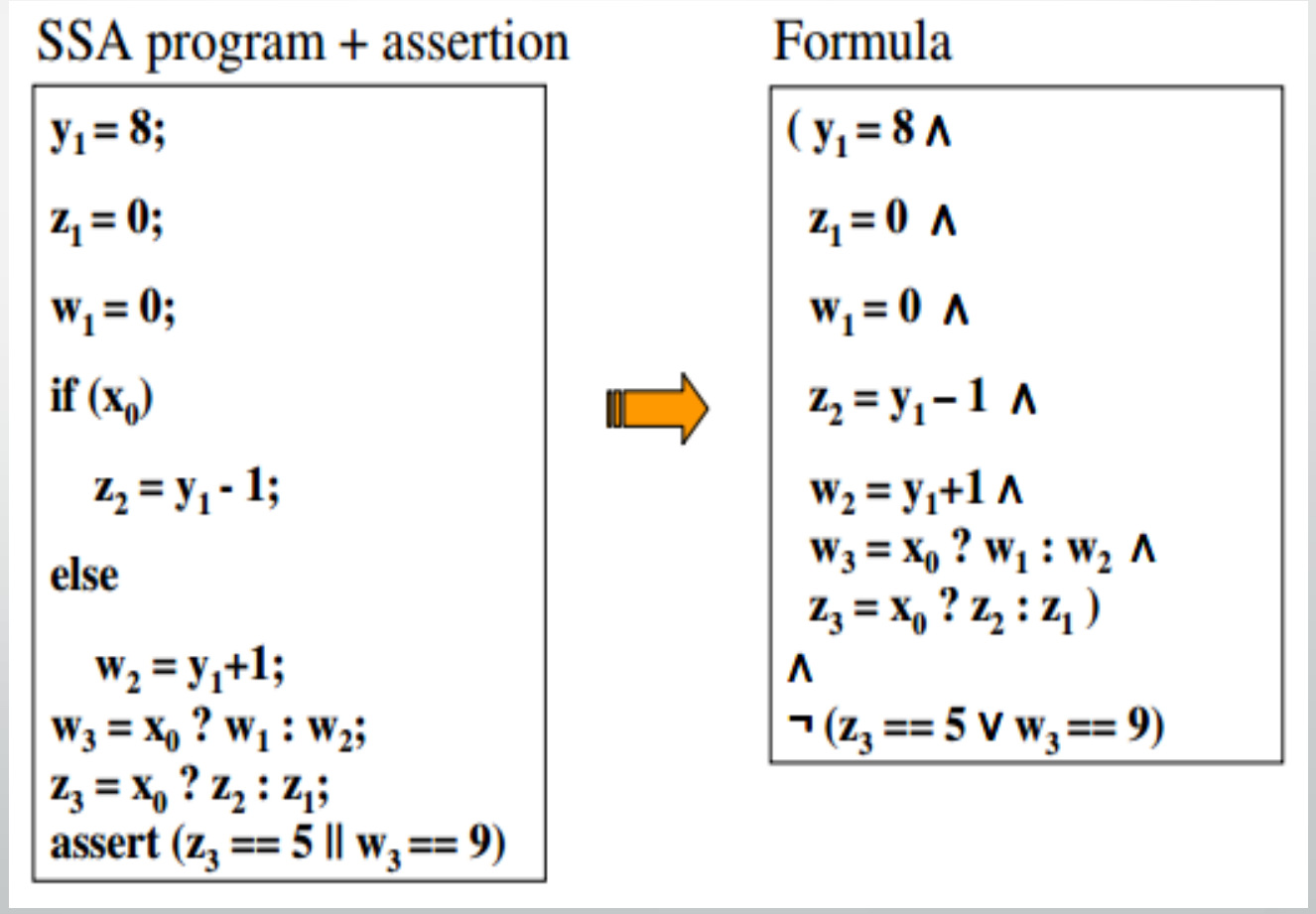
3. Conversione in SSA (2)

- **Soluzione**: usare ulteriori controlli per comprendere il flusso di esecuzione del programma



4. Conversione in un insieme di equazioni

- In questa fase si passa da SSA ad una formula analizzabile dai SAT-solvers;
- La formula è caratterizzata dagli AND tra i seguenti due elementi:
  - Assegnamenti SSA
  - Negazione delle asserzioni



C-BMC

- Usa tutte le trasformazioni viste per il BMC
- Estende il linguaggio di programmazione C con 3 caratteristiche di modellazione:

1. Asserzioni:
   - *Assert(e)* - interrompe un'esecuzione quando *e* è false, altrimenti non fa alcuna operazione

```
void assert (_Bool b) { if (!b)  exit(); }
```

2. Non-determinismo
   - *Nondet_int()* - restituisce un valore intero non-deterministico

```
int nondet_int () { int x; return x; }
```

3. Assunzioni
   - *Assume(e)* - ignora l'esecuzione quando *e* è falsa, altrimenti non fa alcuna operazione

```
void assume (_Bool e) { while (!e) ;  }
```

Esempio assume & assert

```
int array[10]; //defined somewhere
int sum(int k) {
    unsigned i, res;
    res = 0;
    assume (0<= k && k<= 10)
    for (i=0; i < k; i++)
        res+=array[i];
    return res;
}
```

```
int main(int argc, char ** argv) {
    …
    int count;
    …
    assert (0<= count && count <= 10)
    int s = sum(count);

    …
    return 0;
}
```

ensures that the assumption was possible

SMT-based BMC

- Proposto per superare difficoltà sulla verifica software
- **Scopo: creare un tool che traduca in modo preciso le espressioni dei programmi in una formula senza quantificatori applicando un insieme di ottimizzazioni per evitare di sovraccaricare il solver**
- ESW-CBMC: esempio di SMT-based BMC
  - Si basa sul CBMC
  - Usa teorie di verifica SMT
  - Migliore di CBMC e SMT-CBMC

SMT - Satisfiability Modulo Theories

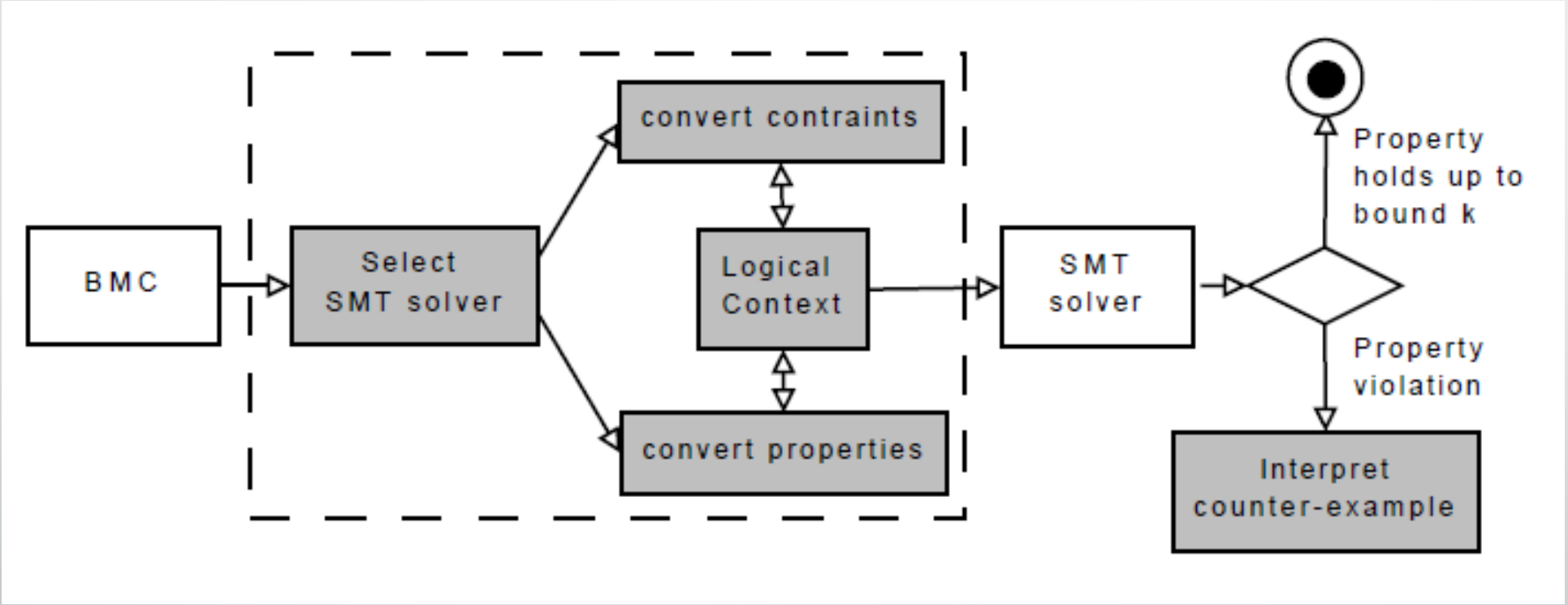
- Data un teoria decidibile T e data una formula senza quantificatori ϕ nello stesso linguaggio di T, ϕ è T-soddisfacibile sse T U {ϕ } è soddisfacibile
- Se la risposta è soddisfacibile allora c'è una violazione; altrimenti no.
- Srotoliamo la transizione di sistema M e la proprietà ϕ fino al bound k, e passiamo ciò al solver SMT per controllare se ci sono violazioni della proprietà ϕ o meno;

$$
\begin{array}{rcl}
Fml & ::= & Fml\ con\ Fml \mid \neg Fml \mid Atom \\
con & ::= & \wedge \mid \vee \mid \oplus \mid \Rightarrow \mid \Leftrightarrow \\
Atom & ::= & Trm\ rel\ Trm \mid Id \mid true \mid false \\
rel & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \\
Trm & ::= & Trm\ op\ Trm \mid Const \mid Id \mid Extract\ [i, j] \\
 & & \mid SignExt\ [k] \mid ZeroExt\ [k] \\
 & & \mid ite\ (Fml,\ Trm, Trm) \\
op & ::= & +_{o,u} \mid -_{o,u} \mid *_{o,u} \mid /_o \mid rem \\
 & & \mid << \mid >> \mid \& \mid \mid \mid \oplus \mid @
\end{array}
$$

Teoria

- *FML* denota espressioni booleane;
- *TRM* denota interi, reali e vettori di bit;
- *OP* rappresenta operazioni binarie;
- *Extract[i,j]* denota estrazioni dal bit i fino al bit j;
- @ denota la concatenazione tra bit;
- *ite(Fml, Trm, Trm)* prende come primo argomento un *Fml* e, in base ai suo valore, seleziona o il secondo o il terzo argomento;
- *select(a, i)* estrae l'elemento *i* dall'array *a;*
- *store(a, i , v)* memorizza l'elemento *v* nell'indice *i* dell'array *a.*

ESW-CBMC


BMC
Select
SMT solver
convert contraints
Logical
Context
convert properties
SMT
solver
Property
holds up to
bound k
Property
violation
Interpret
counter-example

Encoding – Scalar Data Types

- Sono usati 2 approcci per modellare gli integer signed e unsigned:
  - Integer fornito dalla teoria SMT o
  - Vettori di bit
- Sei tipi di dati:
  - Bool
  - Signedbv (rappresenta *int*, *long int*, *long long int*, *char* dell'ANSI-C)
  - Unsignedbv (rappresenta la versione unsigned di *signedbv)*
  - Fixedbv
  - Floatbv
  - Pointer

Encoding - Arithmetic overflow e underflow

- Underflow e overflow sono molto frequenti nello sviluppo di sistemi embedded
- Overflow di tipi integer unsigned (unsigned int, unsigned long int) => funzione modulo ($r \bmod 2^w$)

- Overflow di tipi signed (int, long int) => $overflow^{*,/,-,+}(x,y)$
  - restituisce true sse x *,/,-,+ y è superiore a LONG_MAX
- Underflow di tipi signed (int, long int) => $underflow^{*,/,-,+}(x,y)$
  - restituisce true sse x *,/,-,+ y è inferiore a LONG_MIN

Encoding – Arrays e strutture

- Array codificati in teorie con:
  - WITH => store(a, i, v)
  - [] => select(a, i)
- Strutture codificate in teorie con:
  - Store(w, f, v)
  - Select(w, f)

Puntatori

- ANSI-C prevede 2 operatori di dereferenziazione:
  - *p
  - [i]
- Codificati come una tupla avente 2 campi:
  - P.o.: codifica l'oggetto a cui il puntatore punta;
  - P.i.: codifica un indice all'interno di tale oggetto
- Il VCG genera 2 proprietà correlate alla sicurezza dei puntatori:
  1. Controllare se il puntatore punti ad un oggetto corretto;
     - Un oggetto invalido è codificato come ν
  2. Controllare se il puntatore è NULL;
     - NULL codificato come ή
  - **ή e ν sono utili per controllare puntatori invalidi**

Esempio: INVALID_POINTER =>

$$l_i \Leftrightarrow (p.o \neq \nu) \wedge (p.i \neq \eta)$$

Esempio

1

```
1 int main() {
2 int a[2], i, x;
3 if (x==0)
4   a[i]=0;
5 else
6   a[i+2]=1;
7 assert(a[i+1]==1);
8 }
```

(a)

```
1 g1 |= (x1 == 0)
2 a1 == (a0 WITH [i0:=0])
3 a2 == a0
4 a3 == (a2 WITH [2+i0:=1])
5 a4 == (g1 ? a1 : a3)
6 t1 == (a4[1+i0] == 1)
```

(b)

Figure 2. (a) A C program with violated property. (b) The C program of (a) in SSA form.

2

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\ a_1 := store(a_0, i_0, 0) \\ \wedge\ a_2 := a_0 \\ \wedge\ a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\ a_4 := ite(g_1, a_1, a_3) \end{bmatrix} \quad (1)$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\ 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\ 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\ select(a_4, i_0 + 1) = 1 \end{bmatrix} \quad (2)$$

3

$$\begin{aligned} l_0 &\Leftrightarrow i_0 \geq 0 \\ l_1 &\Leftrightarrow i_0 < 2 \\ &\vdots \\ l_6 &\Leftrightarrow select(a_4, i_0 + 1) = 1 \end{aligned} \quad (3)$$

4

$$\neg P := \neg l_0 \vee \neg l_1 \vee \ldots \vee \neg l_6 \quad (4)$$

5

$$C \wedge \neg P$$

CBMC vs ESW-CBMC

- CBMC non capace di verificare il programma **ffr1ke** e **lms**
- CBMC richiede molto tempo per i programmi **ludcmp, qurt** e **laplace**
- ESW-CBMC più veloce per i programmi **adpcm, exStbHDMI, exStbLED**
  - Risulta più lento solo per il programma **exStbResolution**
- Per gli altri programmi il tempo di verifica sono molto vicini
- Entrambi sono in grado di trovare bug su underflow e overflow
- ESW-CBMC scala meglio quando i problemi riguardano:
  - aritmetica non lineare;
  - operazioni a bit;
  - puntatori ;
  - manipolazioni di matrici

					CBMC						ESW-CBMC					
					Time			#P			Time			#P		
	Module	#L	B	#P	Encoding	Decision Procedure	Total	Passed	Violated	Failed	Encoding	Decision Procedure	Total	Passed	Violated	Failed
1	sensor	603	5	167	2.04	0.002	2.04	167	0	0	1.23	0.02	1.26	167	0	0
2	crc	125	257	18	5.60	0.003	5.60	18	0	0	4.08	0.07	4.16	18	0	0
3	fft1	218	9	72	0.44	0.001	0.44	72	0	0	0.43	0.005	0.43	72	0	0
4	fft1k	155	1025	39	MO	MO	MO	0	0	39	2337.83	0.055	2337.88	39	0	0
5	fibcall	83	30	2	0.19	0	0.19	2	0	0	0.15	0.002	0.15	2	0	0
6	fir	314	34	25	4.88	0.02	4.9	25	0	0	3.36	0.68	4.04	25	0	0
7	insertsort	86	10	17	0.36	0.005	0.37	17	0	0	0.31	0.02	0.32	17	0	0
8	jfdctint	374	65	331	1.22	0.001	1.22	330	1	0	0.45	2.41	2.86	330	1	0
9	lms	258	202	35	MO	MO	MO	0	0	35	132.6	0.24	132.84	35	0	0
10	ludcmp	144	7	88	4.52	TO	TO	87	0	1	0.017	1.44	1.46	88	0	0
11	matmul	81	6	31	1.16	0	1.16	31	0	0	1.06	0.012	1.07	31	0	0
12	qurt	164	20	8	18.83	TO	TO	7	0	1	1.22	7.7	8.92	8	0	0
13	bcnt	86	17	162	4.42	0.05	4.47	162	0	0	1.24	0.89	2.13	162	0	0
14	blit	95	1	129	0.21	0.001	0.21	128	1	0	0.13	0.28	0.41	128	1	0
15	pocsag	521	42	183	15.32	0.1	15.42	182	1	0	12.33	5.77	18.1	182	1	0
16	adpcm	473	100	553	74.34	3.52	77.86	553	0	0	45.73	9.24	54.97	553	0	0
17	laplace	110	11	76	30.81	TO	TO	0	0	76	12.32	0.29	12.62	76	0	0
18	exStbKey	558	20	18	1.23	0.002	1.23	18	0	0	1.22	0.004	1.23	18	0	0
19	exStbHDMI	1045	15	25	167.91	78.97	246.88	25	0	0	164.43	33.53	197.96	25	0	0
20	exStbLED	430	40	6	195.97	129.8	325.77	6	0	0	165.63	44.53	210.16	6	0	0
21	exStbHwAcc	1432	1000	113	0.67	0.002	0.67	113	0	0	0.72	0.004	0.73	113	0	0
22	exStbResolution	353	200	40	271.8	319.13	590.93	40	0	0	269.31	1161.16	1430.47	40	0	0

Table 3. Results of the comparison between CBMC and ESW-CBMC

INTEGRAZIONE

1. Differenza tra invisible formal methods e model checking

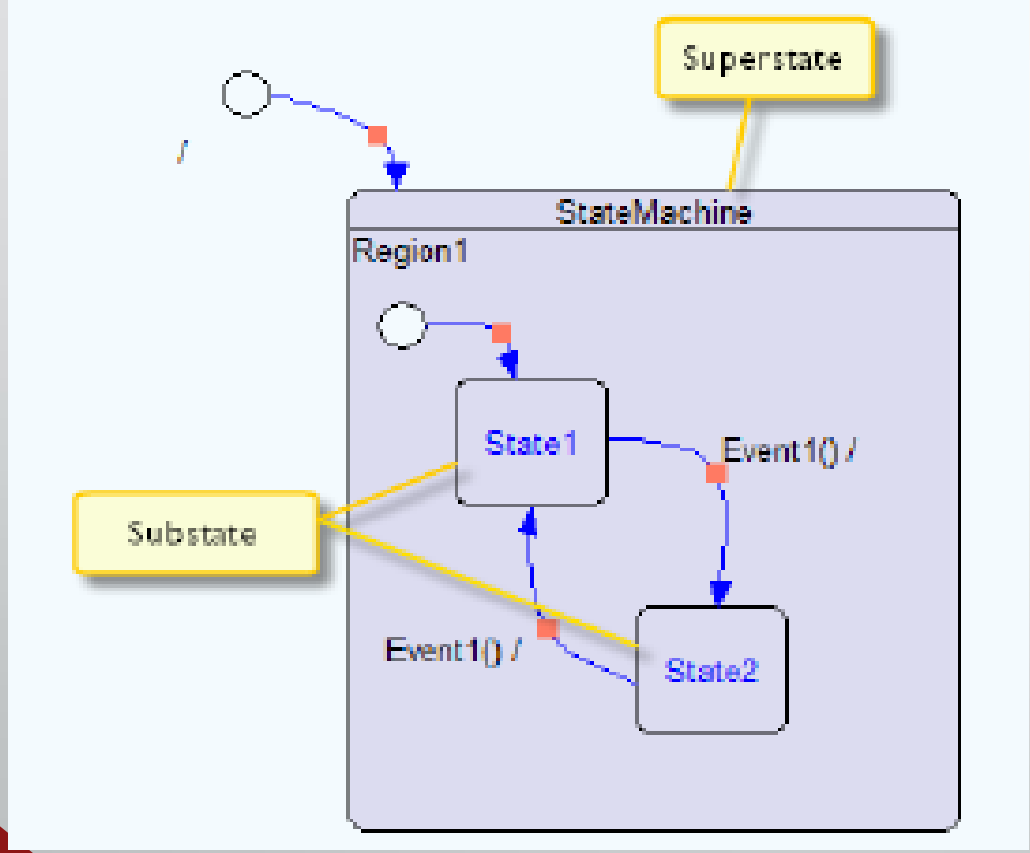
2. Perché usare BMC per i sistemi embedded e non MC?

1. Differenza tra model checking e invisible formal methods

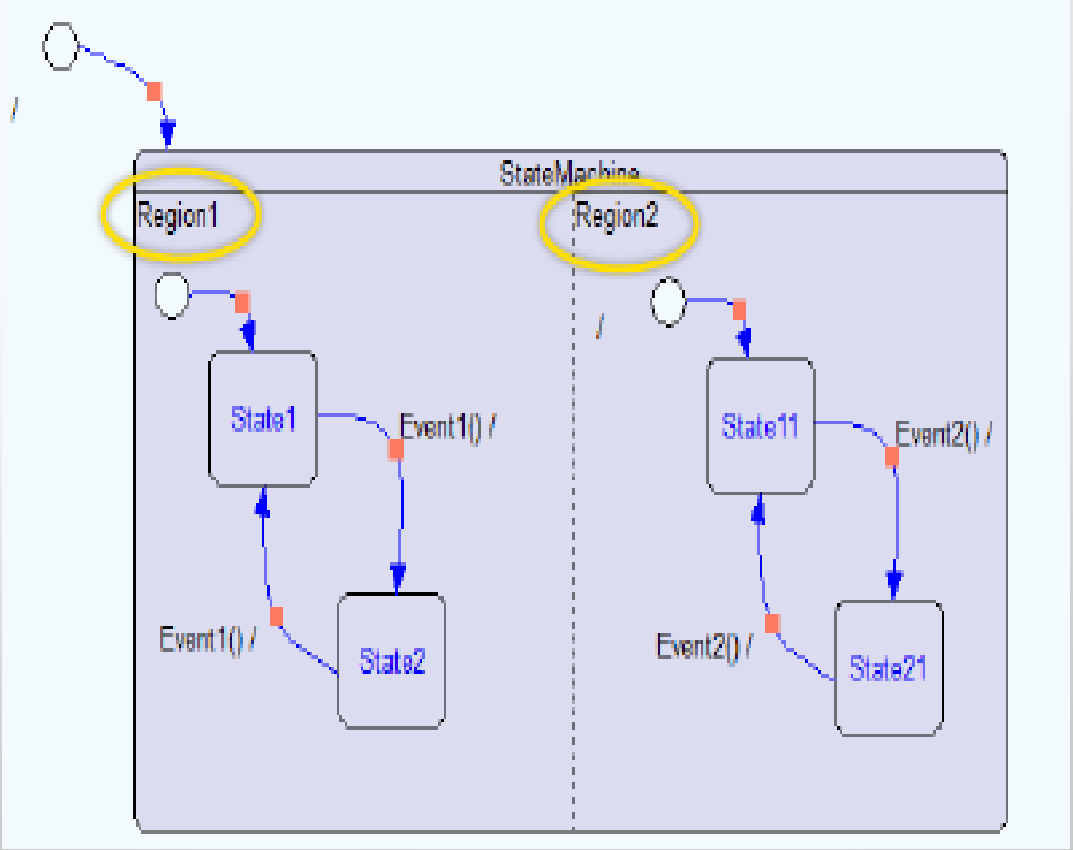
- Model checking
  - Sfrutta automi a stati finiti
- Invisible formal methods
  - Gli algoritmi e la matematica sono nascosti
  - Sfrutta diagrammi UML (stati macchina)
- **Gli stati macchina sono un'evoluzione degli automi capaci di rendere più semplice la descrizione di sistemi complessi**

Diagrammi UML (stati macchina)

Definizione di stati macchina gerarchici



Gestione della concorrenza



2. Perché usare BMC per i sistemi embedded e non MC?

- Il problema del MC per i sistemi embedded è dato dall'esplosione degli stati
- Per sistemi con task concorrenti, il numero complessivo degli stati cresce in modo esponenziale con il numero dei processi coinvolti
- Nel BMC lo spazio degli stati viene generato fino ad una profondità k fissata
  - Limita il numero degli stati complessivi
  - E' possibile usare tecniche **induction-based verification**

Assert vs Assume

- **Una asserzione è inserita dopo l'ultima iterazione. Essa sarà violata se il programma esegue più iterazioni che il bound permette.**

```
void f(...) {
  ...
  if (cond) {
    Body;
    if (cond) {
      Body;
      if (cond) {
        Body;
        assert (!cond);
      }
    }
  }
 ...
}
```

```
void assert (_Bool b) { if (!b)  exit(); }
```

```
void assume (_Bool e) { while (!e) ;  }
```

Riferimenti

1. Sistemi embedded – wikipedia
2. Verification Tools for Embedded Systems
3. Il progetto dei sistemi dedicati (*embedded systems*)
4. Introduction to Embedded Software Verification
5. Model-Based Verification and Validation of Embedded Systems
6. SMT-Based Bounded Model Checking for Embedded ANSI-C Software
7. SMT-Based Context-Bounded Model Checking for Embedded Systems: Challenges and Future Trends
8. Introduction to CBMC
9. http://ftp.iar.se/WWWfiles/vs/UserGuide.pdf

GRAZIE PER L'ATTENZIONE