Generazione Automatica di Test per Analisi di Copertura di Software Safety Critical utilizzando Bounded Model Checking

Massimo Narizzano

j.w.w.

D. Angeletti, **E. Di Rosa**, **E. Giunchiglia**
G. Palma, **A. Puddu**, **S. Sabina**

AnsaldoSTS

11 maggio 2011

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

Ambito Applicativo

Sistemi Safety Critical

Un sistema **safety-critical** è un sistema il cui malfunzionamento può causare morti o infortuni gravi, ingenti perdite di denaro o danni ambientali.

Es: Impianti Nucleari, Aerei, Treni etc..

Il sistema ERTMS/ETCS

Il sistema ERTMS/ETCS

In Europa esistono 20 diversi sistemi di segnalamento ferroviario. ERTMS (*European Rail Traffic Management System*) consente l'interoperabilità sulla rete ferroviaria europea.

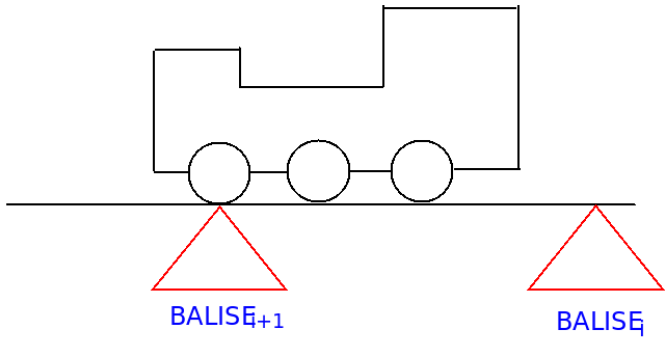
Questo sistema è composto da due elementi:

- il **GSM-R**, il sistema di comunicazione radio basato sullo standard GSM (quello della telefonia mobile), ma che utilizza frequenze diverse, proprie delle ferrovie;
- l'**ETCS** (European Train Control System), che permette di trasmettere al macchinista le informazioni relative alla velocità consentita e di controllarne il rispetto.

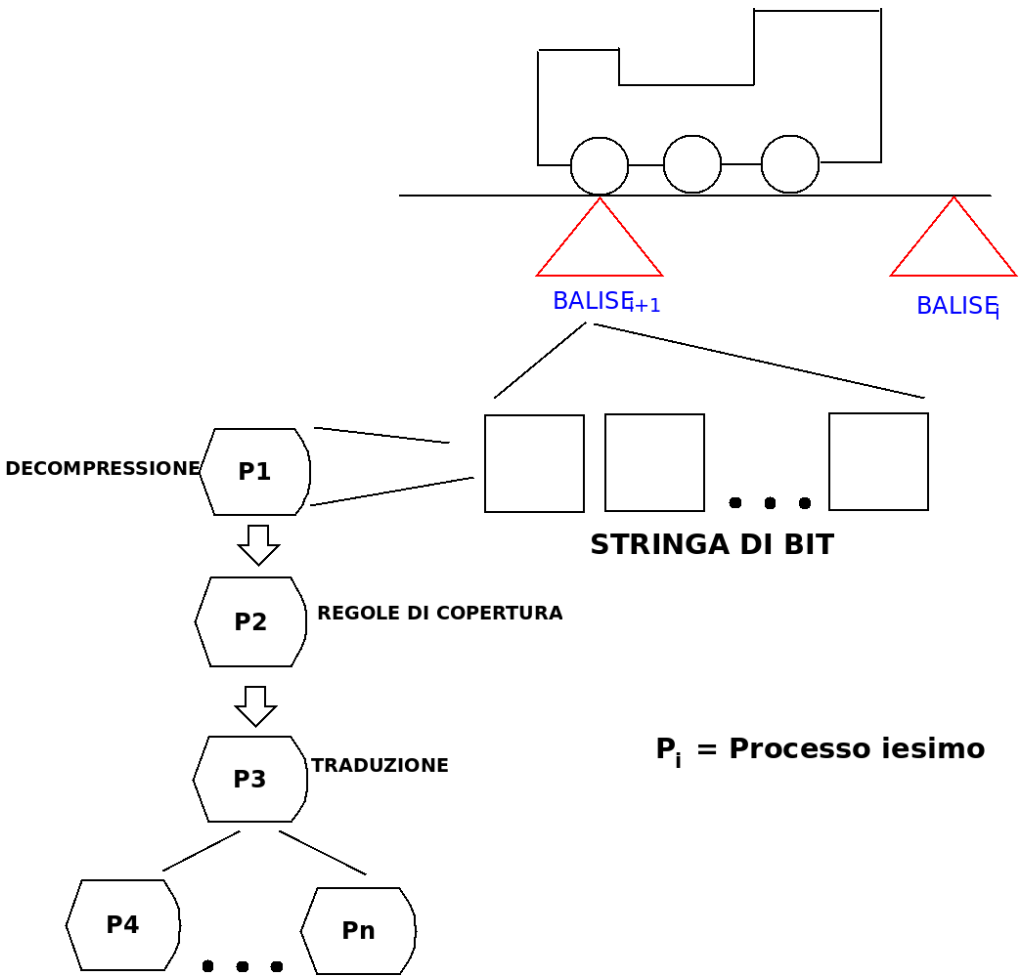
AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

Il sistema ERTMS/ETCS



Il sistema ERTMS/ETCS



Problema

Dati n processi : $P_i = \langle C_i, P_i, D_i \rangle$ con i = 1..n
dove :

- C_i = tempo di *Computazione* del processo i-esimo
- P_i = *Periodicità* del processo i-esimo
- D_i = *Deadline* (strict) entro la quale deve essere eseguito P_i

Schedulazione ottima di n processi entro un tempo massimo (T)

Problemi

- Schedulazione di processi
- Worst Case Execution Time Analysis (C_i = ?)
- Verifica del codice dei processi P_i, secondo le normative Europee CENELEC 50128.

Verifica del Software

Verifica Statica

Verifica Formale: dimostrare formalmente la (in)correttezza

- Possono verificare la totale assenza di errori
- Difficoltà nell'essere usate
- Utilizzano troppe risorse(tempo e memoria)

Verifica Dinamica

Testing Verifica del comportamento del sistema tramite l'esecuzione

- Semplice da applicare e poche risorse usate
- Trova gli errori e aumenta la confidenza nella correttezza
- Costi troppo alti (50%), dovuti alla poca automazione nel processo, in particolare nella **generazione dei test**

Obiettivo

Sviluppo di un insieme di tecniche per la Verifica di Software in ambiente Safety-Critical.

Sviluppo di una procedura di Testing per la riduzione dei costi e che aumenti la confidenza nella correttezza del Software.

Sviluppo di nuove metodologie per l'automazione del processo di generazione di Test.

Outline

Outline

Verifica Formale

Verifica : Assicurare che il sistema ...

- si comporti come desiderato;
- sia senza errori.

Formale:

Utilizzando linguaggi (formali), tecniche e procedure basate sul rigore matematico.

Risultato Desiderato

Una “proof” della (in)correttezza del sistema.

Verifica Formale

- *Model checking*, dato un modello del sistema viene eseguita un'esplorazione esaustiva dello spazio degli stati. state space enumeration, symbolic state space enumeration, abstract interpretation, symbolic simulation, abstraction refinement. Le proprietà da verificare sono spesso definite in logica temporale come Linear Temporal Logic (LTL) oppure Computational Tree Logic (CTL).
- *Logical Inference* Dimostrare che una certa proprietà (congettura), è una valida conseguenza di un insieme di assiomi che descrivono il problema (Theorem Proving).

Model Checking

Scelta una proprietà da verificare, espressa come una formula logica temporale p, e un modello M ottenuto da una struttura di Kripke che descrive tutti i comportamenti del sistema, e dato lo stato iniziale s, decidere se

$$M, s \models p$$

- True, il sistema funziona in accordo con la proprietà
- False, il sistema e' incorretto, viene fornito un controesempio.

PROBLEMA

Esplosione combinatoria degli stati

Bounded Model Checking

- Codifica di tutte le esecuzioni del sistema M di lunghezza K in una formula proposizionale $|[M]|^k$
- Codifica della proprietà nella formula $|[\neg\phi]|^k$, che è vera per tutte le esecuzioni del sistema che violano la proprietà ϕ
- Se la formula $|[M]|^k \wedge |[\neg\phi]|^k$ è soddisfacibile allora la proprietà non è soddisfatta e viene costruito un controesempio.
- Se la formula $|[M]|^k \wedge |[\neg\phi]|^k$ è insoddisfacibile allora non esiste un controesempio di lunghezza k che falsifichi ϕ.

PROBLEMA

BMC trova errori non dimostra la correttezza di sistemi

AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

CBMC

CBMC è un bounded model checker che permette di verificare le seguenti proprietà di un codice scritto in linguaggio *C*:

- *pointer safety*: indirizzi dei puntatori sempre validi
- *array bound*: non si accede mai oltre il limite (sup/inf) di un array
- ***user assertion*** : proprietà definite dall'utente tramite assertion.

Il programma e le proprietà da verificare vengono ricondotte ad una formula booleana che viene passata ad un SAT solver.

Nel caso in cui la verifica di una delle proprietà fallisce viene riportata una **traccia d'errore.**

CBMC

data una funzione

resto_2 (*n*)

```
x=0;
for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
assert(x<2);
return x;
```

CBMC

data una funzione

```
resto_2 ( n )

 x=0;
 for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
 assert(x<2);
 return x;
```

fissato k = 1

```
    resto_2 ( n )
(1) x=0; i=1;
      if i <= n
        if x == 2
(2)       x=0;
        else
(3)       x= x + 1;
(4)
(5) assert(x<2);
(6) return x;
```

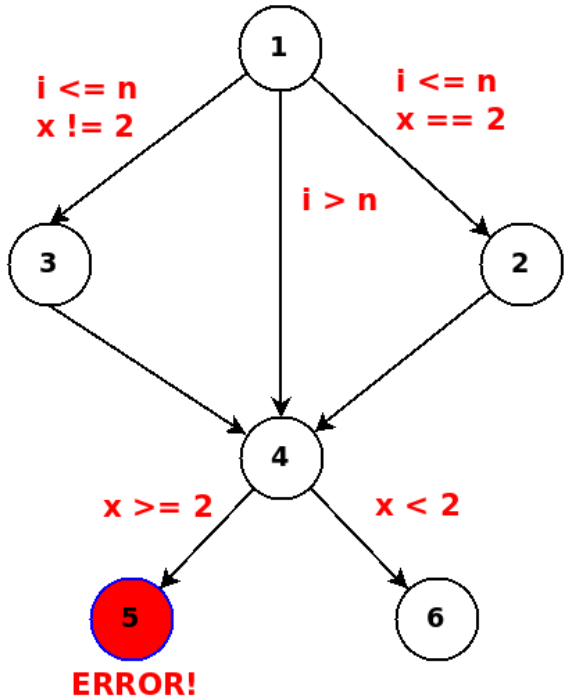
CBMC

data una funzione

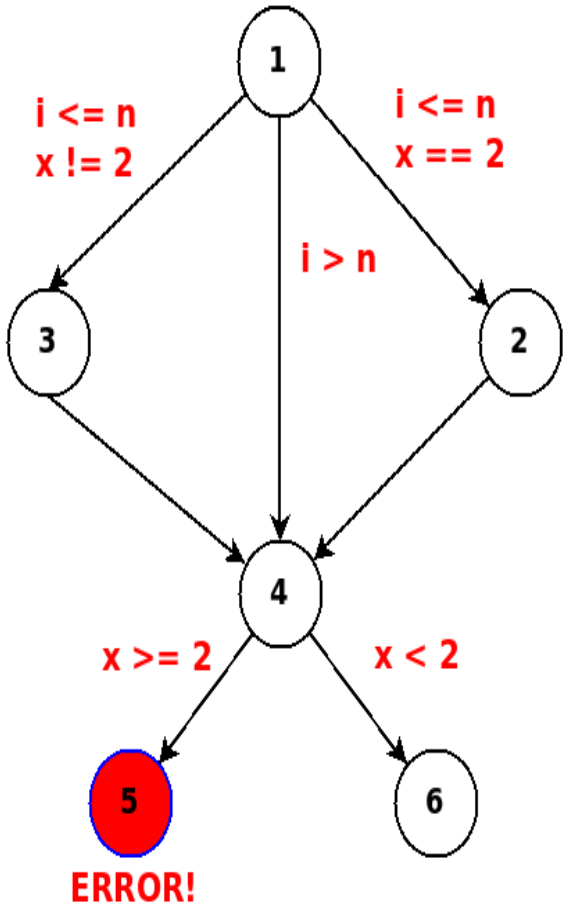
```
resto_2 ( n )

  x=0;
  for (i=1; i<= n;i=i+1)
     if x == 2
       x=0;
     else
       x= x + 1;
  assert(x<2);
  return x;
```

fissato k = 1

```
    resto_2 ( n )
(1)  x=0; i=1;
       if i <= n
         if x == 2
(2)       x=0;
         else
(3)       x= x + 1;
(4)
(5)  assert(x<2);
(6)  return x;
```

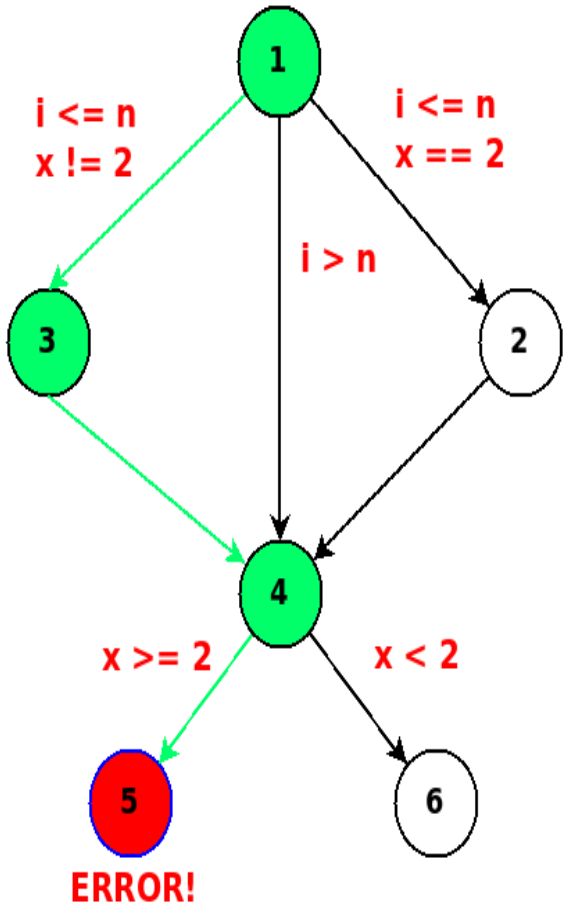


User Assertion



Si basa sul concetto di raggiungibilità degli stati di errore.

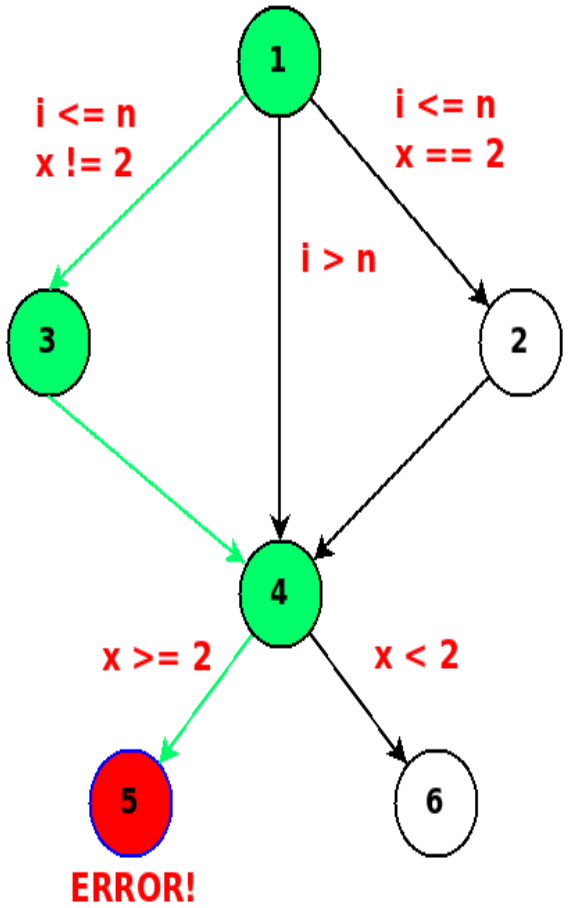
User Assertion



Si basa sul concetto di raggiungibilità degli stati di errore.

- Esiste un percorso che raggiunga lo stato 5?

User Assertion

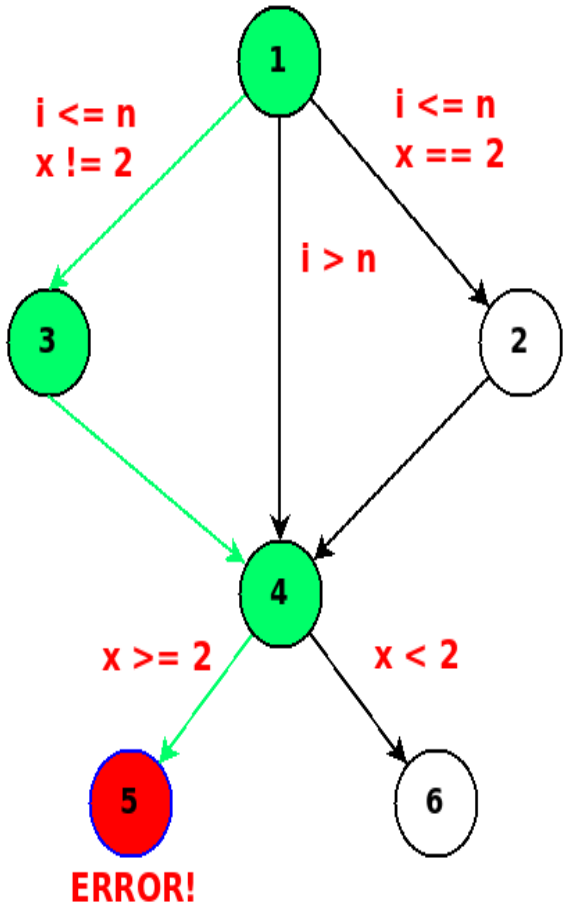


Si basa sul concetto di raggiungibilità degli stati di errore.

- Esiste un percorso che raggiunga lo stato 5?
- trovare i valori per cui il percorso è eseguito e l'assert falsificata.

$$i \leq n \wedge x \neq 2 \wedge x \geq 2$$

User Assertion



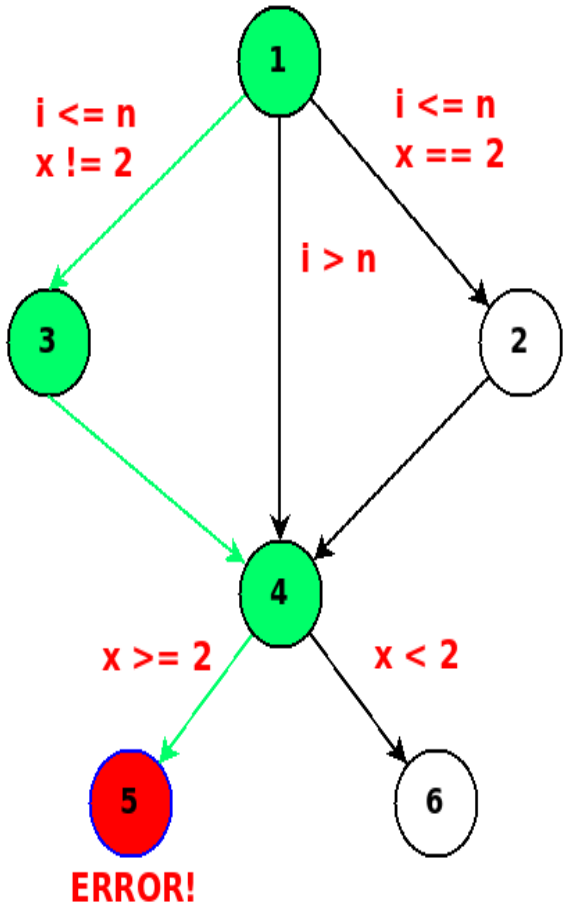
Si basa sul concetto di raggiungibilità degli stati di errore.

- Esiste un percorso che raggiunga lo stato 5?
- trovare i valori per cui il percorso è eseguito e l'assert falsificata.

$$i \leq n \wedge x \neq 2 \wedge x \geq 2$$

$$x = 3 \wedge i = 2 \wedge n = 3$$

User Assertion



Si basa sul concetto di raggiungibilità degli stati di errore.

- Esiste un percorso che raggiunga lo stato 5?
- trovare i valori per cui il percorso è eseguito e l'assert falsificata.

$$i \leq n \wedge x \neq 2 \wedge x \geq 2$$

$$x = 3 \wedge i = 2 \wedge n = 3$$

- Non ci sono soluzioni

CBMC

```
x = 0; i = 1;
if i <= n
   if x == 2
      x=0;
   else
      x= x + 1;
assert(x < 2)
return x;
```

CBMC

```
x = 0; i = 1;
if i <= n
  if x == 2
    x=0;
  else
    x= x + 1;
assert(x < 2)
return x;
```

x_0= 1; i_0= 1;
if(i_0 <= n_0)
  if x_0 == 2
    x_1=0;
  else
    x_2= x_0 + 1;
  x_3= (x_0 == 2) ? x_1 : x_2;
x_4= (i_0 <= n_0) ? x_3 : x_0;
assert(x_4 < 2);

CBMC

```
x = 0; i = 1;
if i <= n
  if x == 2
    x=0;
  else
    x= x + 1;
assert(x < 2)
return x;
```

```
x0= 1; i0= 1;
if(i0 <= n0)
  if x0 == 2
    x1=0;
  else
    x2= x0 + 1;
  x3= (x0 == 2) ? x1 : x2;
x4= (i0 <= n0) ? x3 : x0;
assert(x4 < 2);
```

$$
\begin{aligned}
P:= \quad & x_0 = 1 \wedge i_0 = 1 \wedge \\
& x_1 = 0 \wedge \\
& x_2 = x_0 + 1 \wedge \\
& x_3 = (x_0 == 2) ? x_1 : x_2 \wedge \\
& x_4 = (i_0 <= n_0) ? x_3 : x_0 \\
C:= \quad & x_4 < 2
\end{aligned}
$$

CBMC

```
x = 0; i = 1;
if i <= n
  if x == 2
    x=0;
  else
    x= x + 1;
assert(x < 2)
return x;
```

```
x0= 1; i0= 1;
if(i0 <= n0)
  if x0 == 2
    x1=0;
  else
    x2= x0 + 1;
  x3= (x0 == 2) ? x1 : x2;
x4= (i0 <= n0) ? x3 : x0;
assert(x4 < 2);
```

$$
\begin{aligned}
P:= \quad & x_0 = 1 \wedge i_0 = 1 \wedge \\
& x_1 = 0 \wedge \\
& x_2 = x_0 + 1 \wedge \\
& x_3 = (x_0 == 2) ? x_1 : x_2 \wedge \\
& x_4 = (i_0 <= n_0) ? x_3 : x_0 \\
C:= \quad & x_4 < 2
\end{aligned}
$$

Se esiste un valore n_i: P è vera e C è falsa allora n_i è una traccia di errore. In CBMC $P \wedge \neg C$ viene convertita in una formula booleana la cui soddisfacibilità produrrà un n_i

Outline

Testing

Processo di “verifica” tramite l’esecuzione del software. Dato un processo $P(\underline{X})$ (dove $\underline{X}$ è un vettore di variabili di input) si possono individuare le seguenti fasi

- Generazione (o selezione) di un insieme di valori $\underline{X}$’ (test)
- Esecuzione(simulata o/e concreta) di $P(\underline{X}')$ per ogni $\underline{X}$’
- Collezione e confronto dei risultati con i valori attesi

Generazione/Selezione dell’insieme dei Test

Generazione/Selezione dei Test

Tutti i possibili test?

HP: Dato P con ($|\underline{X}| = 1$) => 65536 valori(test) diversi. Tempo di esecuzione 1 millisecondo per ogni test => Testing in meno di due minuti.
Se $|\underline{X}| = 3$) in poco meno di **9000** anni potremo eseguire gli oltre **280000 miliardi** di test.

Quali test selezionare?

Analisi di Copertura

Creare un insieme di test che esplori tutti i possibili “comportamenti” del sistema secondo determinati criteri.

Criteri di Copertura

- **Statement coverage**: Tutte le righe di un programma devono essere eseguite almeno una volta;
- **Branch Coverage**: ogni predicato di un if-statement deve essere valutato sia a true che a false
- **Condition Coverage**: ogni clausola all'interno di una condizione di un if-statement deve essere valutata almeno una volta sia a true che false.
- **Multiple−condition coverage**: Esplorare tutta la tavola di verità di ogni condizione di un if-statement.
- **Path Coverage**: tutti i path all'interno di un Control Flow Graph del Programma devono essere esplorati

Esempio

Criterio di copertura

Statement Coverage

Codice di esempio

```
void function(int a, int b, int c)
{
            V       V       F
    if ( a > 0 || b < 0 || c == 2 )     V
    {
        ...
    }
    ...
}
```

insieme di test

1. {a=1;b=-1;c=1}

Esempio

Criterio di copertura

Branch Coverage

Codice di esempio

```
void function(int a, int b, int c)
{
            F       F       F
    if ( a > 0 || b < 0 || c == 2 )     F
    {
        ...
    }
    ...
}
```

insieme di test

1. {a=1;b=-1;c=1}
2. {a=-1;b=1;c=1}

Esempio

Criterio di copertura

Condition Coverage

Codice di esempio

```
void function(int a, int b, int c)
{
            V       V       V
    if ( a > 0 || b < 0 || c == 2 )     V
    {
        ...
    }
    ...
}
```

insieme di test

1. {a=1;b=-1;c=1}
2. {a=-1;b=1;c=1}
3. {a=1;b=-1;c=2}

Esempio

Criterio di copertura

Multiple-Condition Coverage

Codice di esempio

```
void function(int a, int b, int c)
{
            F       V       V
    if ( a > 0 || b < 0 || c == 2 )     V
    {
        ...
    }
    ...
}
```

insieme di test

1. {a=1;b=-1;c=1}
2. {a=-1;b=1;c=1}
3. {a=1;b=-1;c=2}
4. {a=-1;b=-1;c=2}

Esempio

Criterio di copertura

Multiple-Condition Coverage

Codice di esempio

```
void function(int a, int b, int c)
{

        if ( a > 0 || b < 0 || c == 2 )          {
        ...
    }
    ...
}
```

insieme di test

1. {a=1;b=-1;c=1}
2. {a=-1;b=1;c=1}
3. {a=1;b=-1;c=2}
4. {a=-1;b=-1;c=2}
5. {a=1;b=1;c=2}
6. {a=1;b=1;c=1}
7. {a=-1;b=-1;c=1}
8. {a=-1;b=1;c=2}

Generazione di Test per Analisi di copertura

Dato un programma P, il processo di generazione consiste:

1. $T = \{\}$
2. $T = T \cup \{t\}$ dove t è un test che copre una porzione del programma non ancora coperta da T
3. Criterio di copertura soddisfatto al 100%T? esci: torna al passo 2

Processo di Generazione?

Viene eseguita **manualmente** da un esperto di Testing con conoscenze di base del codice da Testare.
È la più usata in industria, ma anche la più dispendiosa(50% dei costi).

Esempio

Obiettivo: Trovare un insieme di test per fare Branch Coverage.

```
resto_2 ( n )

          x=0;
(1),(2)   for (i=1; i<= n;i=i+1)
(3),(4)     if x == 2
              x=0;
            else
              x= x + 1;
          assert(x<2);
          return x;
```

$T = \{\}$

n = 1 copre i branch (1),(2) e (3)

$T = \{(n = 1)\}$

n = 2 non copre nulla di nuovo

n = 3 copre anche il branch (4).

$T = \{(n = 1), (n = 3)\}$

Outline

Obiettivo

AUTOMAZIONE DEL PROCESSO DI TESTING
Coverage Analysis

Diverse Tecniche

Random, test generati casualmente fino a che non si raggiunge la copertura totale, o con timeout

Path Coverage Analysis, si costruisce il CFG dal quale si estraggono i path per poi generare il vincoli associati al path che risolti generano un test.

ATG:Random

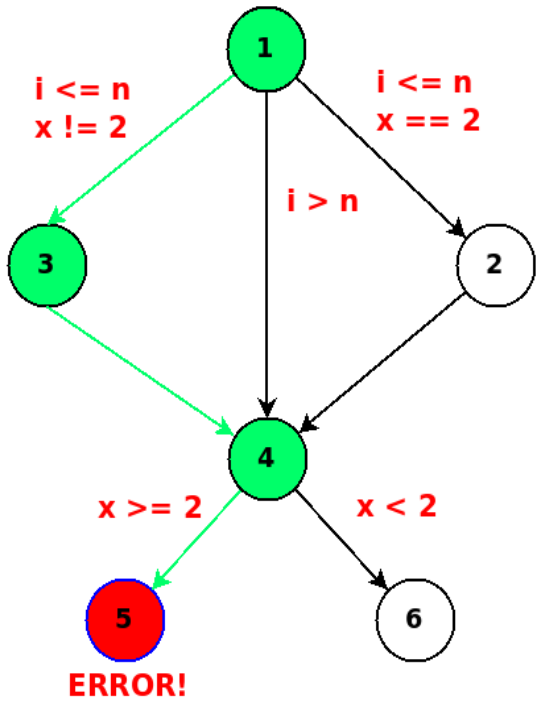
- Genera stringhe di bit a random
- È il modo più veloce e semplice per generare test

```
1 if (x == 2) then
2       x = 0;
3 else
4       x = x + 1;
```

- HP: max int = $N = 16$
- l'istruzione x = 0 ha una probabilità $p = 1/N$ di essere eseguita
- Probabilità che è minore dello **0.002**%.

Difficilmente verranno esplorate parti di codice poco probabili.

Path Coverage Analysis



- Si sceglie un percorso (1),(3),(4),(5)
- Si prendono i predicati associati al percorso
- $$i \leq n \wedge x \neq 2 \wedge x \geq 2$$
- risolvendo i vincoli rispetto a n

User Assertion Violation di CBMC

Analisi di Copertura tramite CBMC: idea

data una funzione

```
resto_2 ( n )   x=0;
  for (i=1; i<= n;i=i+1)
      if x == 2
        assert(0);
        x=0;
      else
        x= x + 1;

  return x;
```

- Modificare CBMC per generare test che garantiscano la copertura
- User Assertion Violation aggiungendo **assert(0)** nella parte di codice che si vuole raggiungere
- CBMC ritorna un valore per le variabili in ingresso che violano la proprietà e coprono quella porzione di codice
- Se non ritorna valore (a K fissato) vuol dire che quel pezzo di codice non può essere raggiunto

AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

Naïve

data una funzione

resto_2 (*n*)

```
x=0;
for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
return x;
```

Naïve

data una funzione

```
resto_2 ( n )

 x=0;
 for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
 return x;
```

fissato k = 1

```
resto_2 ( n )
 x=0; i=1;
 if i <= n
   assert_a(0)
   if x == 2
     x=0;
     assert_b(0)
   else
     x= x + 1;
     assert_c(0)
 else
   assert_d(0)
return x;
```

Naïve

data una funzione

```
resto_2 ( n )

 x=0;
 for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
 return x;
```

fissato k = 1

```
resto_2 ( n )
 x=0; i=1;
 if i <= n
   assert_a(0)
    if x == 2
      x=0;
      assert_b(0)
    else
      x= x + 1;
      assert_c(0)
 else
   assert_d(0)
return x;
```

Risultato

- con K = 1 genero 3 test coprendo (a,c,d) non b
- con K = 3 genero 1 test che copre $assert_b$
- Si genera un test per assert

Naïve

data una funzione

```
resto_2 ( n )

 x=0;
 for (i=1; i<= n;i=i+1)
    if x == 2
      x=0;
    else
      x= x + 1;
 return x;
```

fissato k = 1

```
resto_2 ( n )
 x=0; i=1;
 if i <= n
   assert_a(0)
    if x == 2
      x=0;
      assert_b(0)
    else
      x= x + 1;
      assert_c(0)
 else
   assert_d(0)
return x;
```

Risultato

- con K = 1 genero 3 test coprendo (a,c,d) non b
- con K = 3 genero 1 test che copre $assert_b$
- Si genera un test per assert

Più test del necessario

Per copertura al 100% basta un test (n = 3) a K = 3.

AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

Test Ottimale

La procedura Naïve è semplice, ma genera troppi test.

Test Set Minimo

T è *Minimo*, se non esiste *T'* tale che $|T'| < |T|$

Calcolare Tutti i Test possibili, impossibile in pratica

Test Set Minimale

T è Minimale, se non esiste $T' \subset T$ tale che $|T'| < |T|$ (non ci sono test rindondanti)

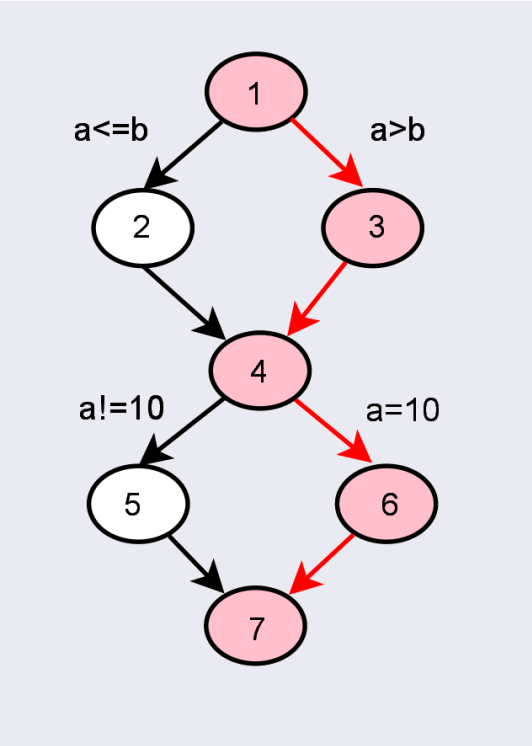
Calcolo A posteriori

Due euristiche per computare insiemi di test Minimali Forward

TeGeVe

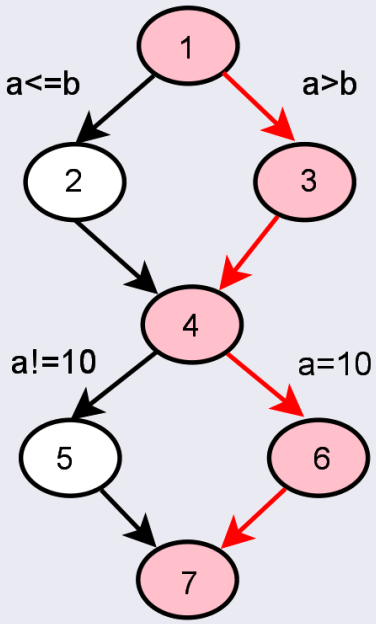
- Aggiungiamo un solo assert alla fine del programma e guidiamo CBMC con un path ben preciso tramite il costrutto **assume(cond)**
- Ogni path forzato deve coprire almeno un branch non coperto da nessun test già generato
- Le parti di codice che non sono nel path vengono scartate e non tradotte in formule booleane, riducendo di fatto la dimensione della formula booleana.
- Ogni volta che un path è unfeasible faccio backtracking e scelgo un nuovo path.

TeGeVe



- Esecuzione forzata di un determinato path (**assume**)
- Se è eseguibile, CBMC ritorna la traccia d'errore(test)
- Si cerca un nuovo path che contenga quanti più nodi non visitati possibile.
- Se non è eseguibile si fa backtracking su un nuovo path.

TeGeVe



- Esecuzione forzata di un determinato path (**assume**)
- Se è eseguibile, CBMC ritorna la traccia d'errore(test)
- Si cerca un nuovo path che contenga quanti più nodi non visitati possibile.
- Se non è eseguibile si fa backtracking su un nuovo path.

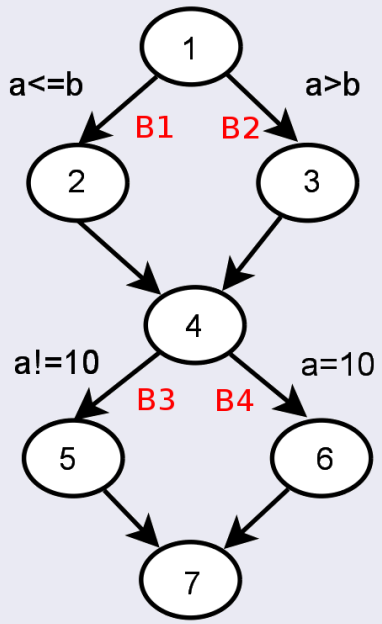
Problema

Rallenta se ci sono molti path unfeasible.

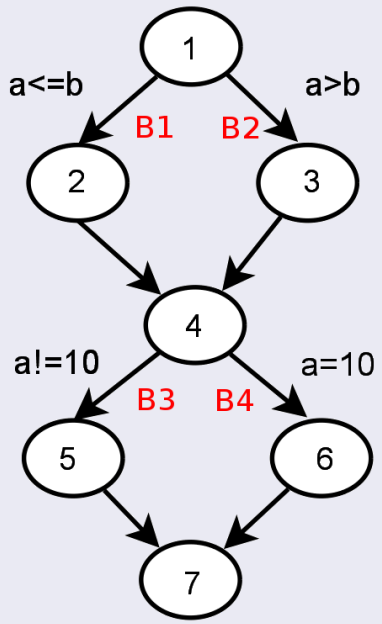
AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

SAT&PREF

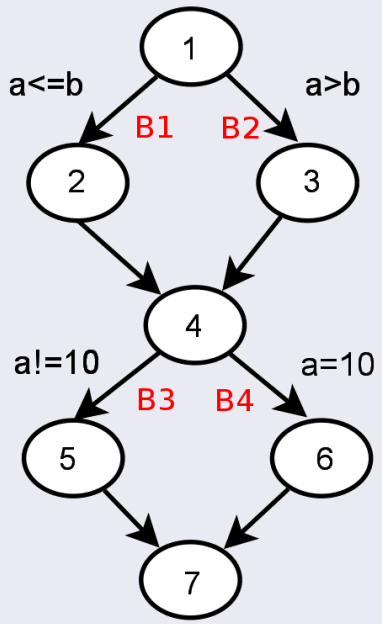


SAT&PREF



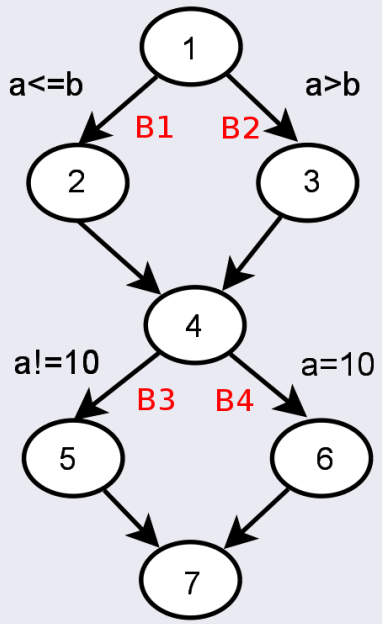
- Aggiungere una variabile per ogni branch B1, B2, B3, B4

SAT&PREF



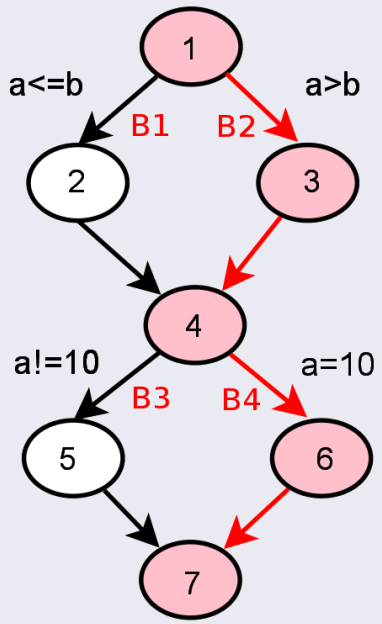
- Aggiungere una variabile per ogni branch B1, B2, B3, B4
- Insieme di Preferenze(*Pref*) con le variabili aggiunte(B1, B2, B3, B4)

SAT&PREF



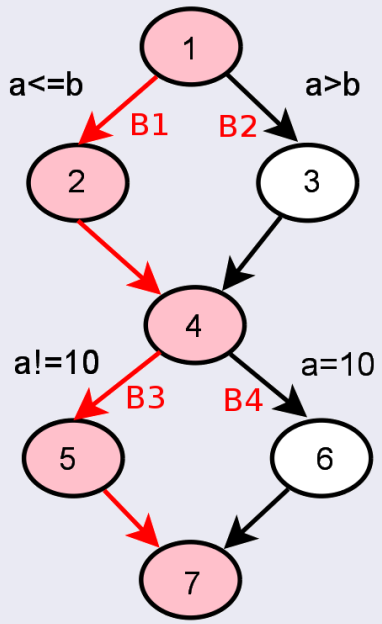
- Aggiungere una variabile per ogni branch B1, B2, B3, B4
- Insieme di Preferenze(*Pref*) con le variabili aggiunte(B1, B2, B3, B4)
- Modifica del SAT solver che massimizzi il numero di variabili *Pref*

SAT&PREF



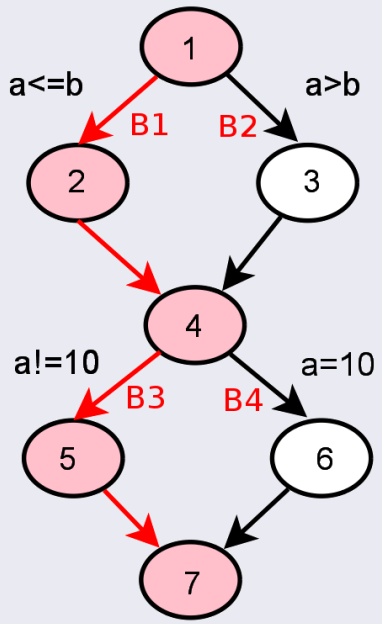
- Aggiungere una variabile per ogni branch B1, B2, B3, B4
- Insieme di Preferenze(*Pref*) con le variabili aggiunte(B1, B2, B3, B4)
- Modifica del SAT solver che massimizzi il numero di variabili *Pref*
- Trovare una soluzione/test che soddisfa/copre il più possibile le preferenze/branch

SAT&PREF



- Aggiungere una variabile per ogni branch B1, B2, B3, B4
- Insieme di Preferenze(*Pref*) con le variabili aggiunte(B1, B2, B3, B4)
- Modifica del SAT solver che massimizzi il numero di variabili *Pref*
- Trovare una soluzione/test che soddisfa/copre il più possibile le preferenze/branch
- Eliminare da *Pref* le preferenze già esplorate e rilanciare SAT solver

SAT&PREF



- Aggiungere una variabile per ogni branch B1, B2, B3, B4
- Insieme di Preferenze(*Pref*) con le variabili aggiunte(B1, B2, B3, B4)
- Modifica del SAT solver che massimizzi il numero di variabili *Pref*
- Trovare una soluzione/test che soddisfa/copre il più possibile le preferenze/branch
- Eliminare da *Pref* le preferenze già esplorate e rilanciare SAT solver
- Termina quando *Pref* è vuoto

Confronto

- *Naïve*, lavora sul codice C originale, semplice, esegue CBMC una volta per test, ma genera più test del necessario. Minimizzazione necessaria a valle.
- *TeGeVe*, modifica il codice C a bound fissato, crea formule booleane più piccole, esegue CBMC un numero di volte maggiore del numero dei test generati, dipende dal numero dei path unfeasible.
- SAT&PREF, modifica direttamente della formula booleana, esegue CBMC una volta per test, crea formule SAT più grandi rispetto a TeGeVe.

Outline

Setup

- Ubuntu con kernel 2.4.27-2, RAM 4GB, 2.0 GHz
- Parte dell'ERTMS (31 Moduli)
- Open Source Benchmarks
- Circa 1000 linee di C e 15 funzioni a modulo
- Ansaldo ha stimato15 minuti per ogni test generato
- 4 metodologie : Manuale più le tre presentate.
- 1 Tool esterno CUTE (concolic unit testing)

Analisi sperimentale - software *safety critical*

Bench Stat		Manual	Naïve			TeGeVe			SAT&PREF		
Module	#F	#T	Time	#T	$\#T_M$	Time	#T	$\#T_M$	Time	#T	$\#T_M$
M_01	27	127	10236	153	113	4391	107	107	10444	105	105
M_02	12	57	2919	149	47	2167	40	40	1076	40	40
M_03	1	17	16	12	6	8	6	6	13	6	6
M_04	1	43	1075	83	48	1265	41	41	845	41	41
M_06	8	39	2452	123	46	10341	27	27	1521	27	27
M_08	24	200	6227	280	158	2504	149	148	3621	146	145
M_09	16	144	2117	182	111	1498	108	108	2514	108	108
M_10	9	35	11017	98	39	1385	41	40	2727	38	38
M_11	22	312	3733	342	268	2771	263	263	5245	263	263
M_12	12	34	299	35	32	264	32	32	636	32	32
M_13	11	181	2174	210	156	1326	156	156	2973	156	156
M_14	11	157	1883	196	132	1152	132	132	2416	132	132
M_15	30	322	3309	336	257	2370	238	238	4773	238	238
M_16	14	69	901	96	40	369	29	29	756	30	29
M_17	7	36	737	73	28	418	28	28	557	28	28
M_18	27	101	2985	167	84	7223	84	83	3641	83	82
M_19	8	62	188	89	35	87	38	38	158	33	33
M_20	25	119	874	169	110	469	110	110	651	110	110
TOTAL	265	2100	53142	2793	1710	40009	1629	1626	44570	1616	1613

Analisi sperimentale - software *safety critical*

- **Meno Tempo**: circa 98% di tempo in meno (525(Manuale), 15(Naïve), 11(TeGeVe), 12(SAT&PREF))
- **Meno Test**: dal 19%(Naïve) al - 23% di SAT&PREF
- SAT&PREF crea **test set più piccoli** e quasi sempre minimali.
- SAT&PREF copre al 100% **un modulo** in più
- Su 9 moduli non si raggiunge il 100% perchè contengono **codice unreachable** (verificati a mano)
- **Copertura Parziale** (non il 100%) su 3 (4) moduli SAT&PREF(TeGeVe)

AnsaldoSTS

STAR SYSTEMS AND TECHNOLOGIES FOR AUTOMATED REASONING

Analisi sperimentale - codice open source

Bench Stats			TeGeVe				CUTE			SAT&PREF		
Module	#F	#B	Time	Time-PG	#T	%C	Time	#T	%C	Time	#T	%C
fdct	2	4	4212	<1	2	100	<1	2	100	769.25	2	100
fibcall	2	2	<1	<1	2	100	<1	31	100	<1	1	100
fir	2	8	13	<1	5	100	58	730	<100	14.29	2	100
matmult	6	10	M	<1	-	<100	<1	6	100	M	-	<100
prime	5	6	<1	<1	7	100	<1	7	<100	1.2	6	100
qurt	4	16	5252	<1	7	100	<1	4	<100	T	2	<100
sqrt	1	8	1923	<1	3	100	<1	1	<100	771.9	2	100
ud	2	26	1311	<1	2	100	<1	2	-	91.87	2	100

Tabella: Open source WCET benchmarks from
`http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Outline

Conclusioni

- Sviluppato una nuova metodologia per la Generazione Automatica di Casi di Test
- Implementato tre diversi approcci
- Applicato questi approcci a un caso di studio Reale L'ERTMS
- **Riduzione drastica dei costi** associati alla fase di testing, tempo totale ridotto al 2%.
- **Riduzione del numero di test**, aumentando la qualità dei test set
- Confrontato la nostra metodologia su casi di studio open source con un sistema allo stato dell'arte.

Preferenze qualitative su letterali

Preferenze qualitative su letterali

Sono rappresentabili dalla coppia $(S, \prec)$ dove S è un insieme di letterali e $\prec$ è un partial order su di essi (relazione binaria, irriflessiva e transitiva su S).

$\mu \prec \mu'$ (μ *è preferito a* μ')

μ e μ' sono totali assegnamenti. $\mu \prec \mu'$ se e solo se

1. esiste un letterale $l \in S$ con $l \in \mu$ e $\bar{l} \in \mu'$; e
2. $\forall l' \in S \cap (\mu' \setminus \mu)$, $\exists l \in S \cap (\mu \setminus \mu')$ tale che $l \prec l'$.

μ' si dice *dominato da* μ rispetto a $(S, \prec)$.

Soluzione *ottimale*

Una soluzione μ si dice *ottimale* se non esiste un'altra soluzione μ' tale che $\mu' \prec \mu$.

Modellazione delle preferenze

Possiamo esprimere:

- preferenze qualitative su letterali (preferenze bipolari)
- preferenze qualitative su formule Booleane (preferenze condizionali)
- preferenze quantitative su letterali/formule
- il mix di preferenze qualitative e quantitative