Boolean Satisfaction - SAT

Among all possible finite domains, the Booleans is a specially interesting case, where all variables take values 0/1.

In Computer Science and Engineering the importance of this domain is obvious: ultimately, all programs are compiled into machine code, i.e. to specifications coded in bits, to be handled by some processor.

More pragmatically, a vast number of problems may be naturally specified through a set of boolean constraints, coded in a variety of forms.

Among these forms, a quite useful one is the clausal form, which corresponds to the Conjunctive Normal Form (CNF) of any Boolean function.

In such cases, Boolean SATisfiability is often referred to as **SAT.**

SAT Modelling

Example_1: n-queens

This is the classic problem that aims at placing n queens in an n*n chess board so that no two queens attack each other.

Associating the presence of a queen in any of the n^2 positions with a 0/1 variable, the problem is modelled through two types of constraints:

- There must be at least a queen in some sets of positions (rows and columns)
- There must be at most one queen in some sets of positions (rows, columns and diagonals)

Such constraints are easily coded in clausal form.

SAT Modelling

Example_1: n-queens

R1: There must be at least a queen in the set of positions Q_1 to Q_n

- 1 single n-ary clause:

$$Q_1 \vee Q_2 \vee \ldots \vee Q_n$$

R2: There must be at most one queen in the set of positions Q_1 to Q_n

- Several (n*(n-1)/2) binary clauses are required :

$$\neg Q_1 \vee \neg Q_2 ,$$

$$\neg Q_1 \vee \neg Q_3 \ ..$$

$$\neg Q_1 \vee \neg Q_n$$

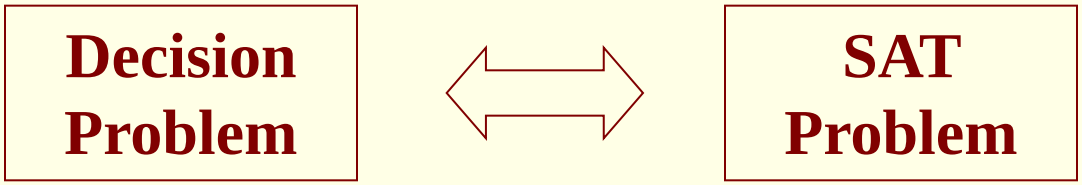
$$\ldots$$

$$\neg Q_{n-1} \vee \neg Q_n$$

Importance of SAT

As can be easily seen, many decision problems can be modelled as Boolean satisfaction problems (SAT).

In fact, “all” decision problems in finite domains may be modelled as a Boolean satisfaction problem



Hence, a general algorithm to solve this type of SAT problems, would solve any decision problem.

Of course, one would only be interested in having an “eficient” algorithm for SAT.

Advanced Techniques in SAT Solvers

- Advanced SAT solvers use techniques common to other Finite Domains solvers, namely
  - (boolean) constraint propagation
  - Heuristics to select the next variable and value to select, so that search is guided towards most promising regions of the search space.
- The specificity of SAT, enables specialised solvers to use additional advanced techniques, not commonly used in more general FD solvers, namely
  - Diagnosis of failure
  - Non-chronological backtracking
  - Learning of “nogood” clauses

Advanced Techniques in SAT Solvers

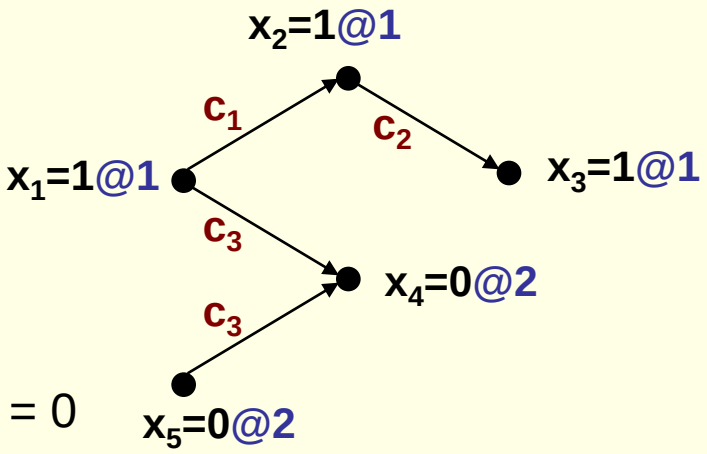
- To illustrate these techniques, we should consider the assignment of values to variables are made. Two different situations occur:
  - Some assignments are explicit decisions made by the solver, selecting the variable and the value.
  - Other assignments are due to propagation on the former.

- For example, take the following labeling on these 3 clauses

$c_1 : (\neg x_1 \vee x_2)$

$c_2 : (\neg x_2 \vee x_3)$

$c_3 : (\neg x_1 \vee \neg x_4 \vee x_5)$

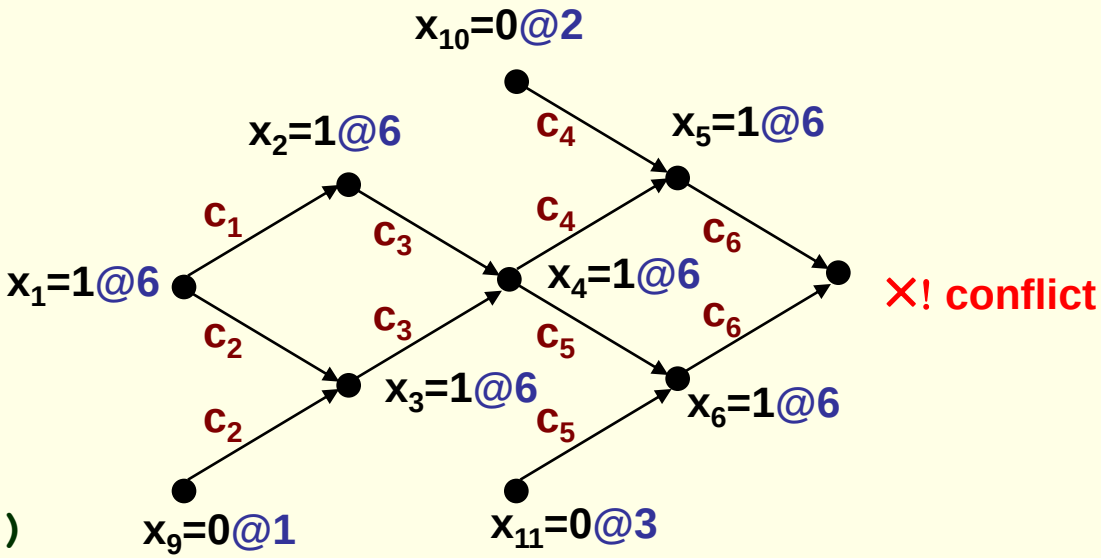


1. A first decision (at level 1) makes $x_1 = 1$
2. Propagation enforces $x_2 = 1$ and $x_3 = 1$
3. A second decision (at level 2) makes $x_5 = 0$
4. Propagation enforces $x_4 = 0$

Diagnosis of Failures

- By maintaining such graph it is easy to detect the real causes of the failures, as illustrated in the graph below.

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

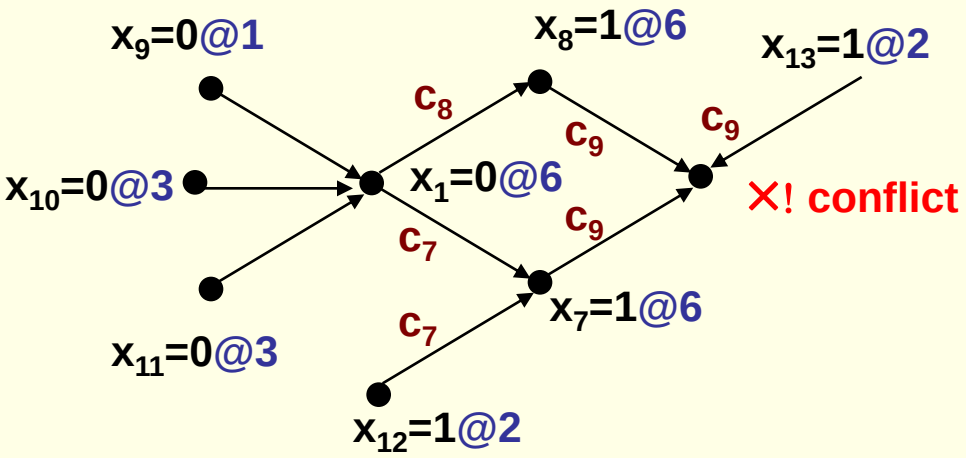


- Clearly, the conflict has been caused by assignments x_1=1, x_9=0, x_{10}=0 and x_{11} = 0, although it was detected in clause c6, envolving variables x_4 and x_5.

Non-chronological Backtracking

- Carrying on with the previous example, x_1 must be set to 0 at level 6.

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...



- Detecting the conflict at level 6, since the causes of the conflict are at decisions taken at levels 1 (X_9), 2 (X_{12} and X_{13}) and 3 (X_{10} and X_{11}) , backtracking is made to decision 3, rather than to decision 5 (and susequently 4) as chronological backtracking would do.

Learning Nogood Clauses

Since, the conflict has been caused by assignments x_1=1, x_9=0, x_{10}=0 and x_{11} = 0, then one may add the clause

$$c_0 : (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$$

to prevent repetition of this impossible assignment.

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

In fact, one may notice that this clause could have been obtained throug resolution on clauses

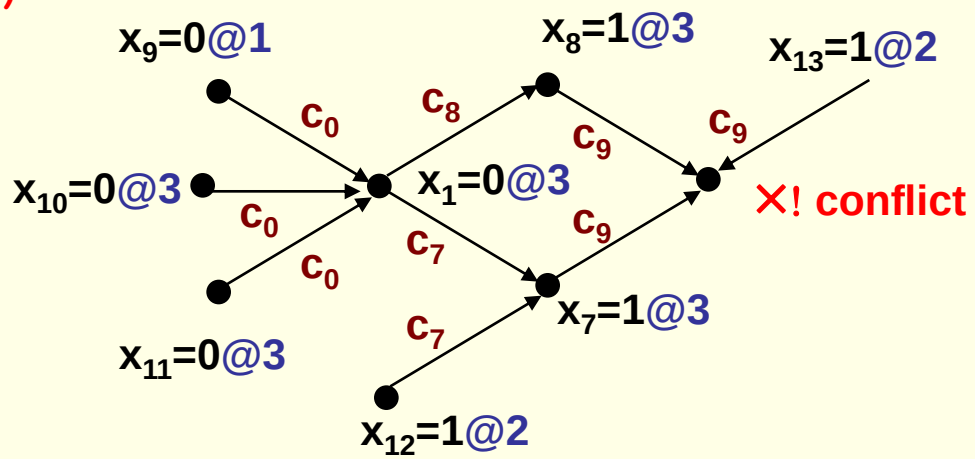
c_1 & $c_3 : (\neg x_1 \vee \neg x_3 \vee x_4)$
& $c_4 : (\neg x_1 \vee \neg x_3 \vee x_5 \vee x_{10})$
& $c_6 : (\neg x_1 \vee \neg x_3 \vee \neg x_6 \vee x_{10})$
& $c_5 : (\neg x_1 \vee \neg x_3 \vee \neg x_4 \vee x_{10} \vee x_{11})$
& $c_3 : (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_{10} \vee x_{11})$
& $c_1 : (\neg x_1 \vee \neg x_3 \vee x_{10} \vee x_{11})$
& $c_2 : (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$

But... how could one guess?

Antecipating Backtracking

- Nogood clauses may also aid to antecipate backtracking. In the previous example, the learned clause sets that $x_{1\ to}$ 0 at level 3.

$c_0 : (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$
$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...



- Instead of detecting the conflict at level 6, by impossibility of assigning a value to variable x1, the cause of the conflict is detected at level 3; since the decisions taken at levels 4 and 5 had no influence on the graph.

SAT solvers and models

- Of course, things are not so simple.
- Not all learned clauses are useful. SAT solvers are ususally paramaterised in order to determine which learned clauses are to be maintained.
- The overhead for carrying on with the analysis for non chronological backtracking might not pay off (which may depend on the heuristics used for variable/value selection). Again parameterisation may help.
- The tuning of all these parameters may be very difficult, and may differ considerably for aparently similar problems.
- Nevertheless current solvers may handle benchmark instances with tens of millions of clauses on around one million variables (not random instances).

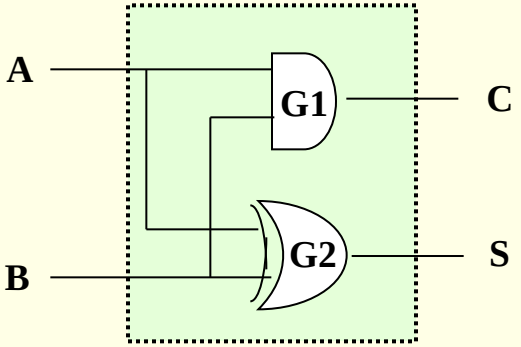
SAT solvers and models

- The critics will say
  - These numbers are misleading. Much less variables and constraints are required if problems are modelled with FD constraints.
  - Processing nogoods is simply learning a structured model that was destroyed when encoding the problem into the “poorly expressive ” SAT clauses.

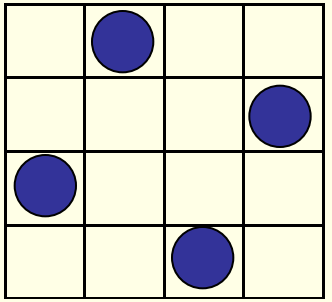
- Despite criticisms, it is clear that SAT solving is quite interesting, having shown great results, and offers possibilities for hybridization with FD solvers.

Restrições Boleanas

- O domínio dos boleanos (ou variáveis 0/1) tem especial aplicação em aplicações
  - Envolvendo circuitos digitais
    - Exemplo: Circuito semi-somador



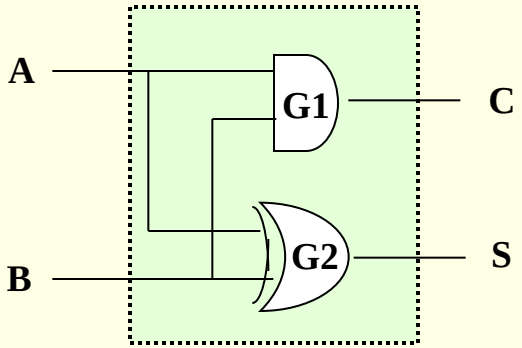
  - Em problemas envolvendo escolhas binárias
    - Exemplo: Rainhas



  - Em problemas que envolvam conjuntos

Restrições Boleanas

- Nas restrições boleanas (de igualdade) podem ser utilizadas os habituais operadores (*not, and, or, nand, nor, xor*, ...).



Modelo do semi-somador

```
C = and(A,B)
S = xor(A,B)
```

Restrições Boleanas

- As restrições (correspondentes às variáveis boleanas) podem ser igualmente expressas com esses operadores

Q1	**Q2**	**Q3**	**Q4**
Q5	**Q6**	**Q7**	**Q8**
Q9	**Q10**	**Q11**	**Q12**
Q13	**Q14**	**Q15**	**Q16**

Modelo das 4-rainhas

```
or(Q1, Q2, Q3, Q4)   % Linha 1
and(Q1,Q2) = 0       % Linha 1
                ....
and(Q1, Q6) = 0      % Diagonal
```

Restrições Boleanas

- A satisfação de restrições boleanas pode ser abordada de várias formas diferentes

  - **Simbolicamente**
    - Unificação boleana

  - **SAT**
    - Colocação de todas as restrições na forma clausal
    - Resolução construtiva (retrocesso) ou reparativa (pesquisa local)

  - **Domínios finitos**
    - O domínio 0/1 é um domínio finito com 2 valores
    - Resolução comum aos domínios finitos

Restrições Booleanas

- Para verificar a satisfação de restrições boleanas de uma forma simbólica é conveniente converter todas as restrições de forma a usar apenas,

  – os operadores

    + (ou-exclusivo) e **·** (conjunção),

  – constantes boleanas

    0 e **1** (e outras constantes, dependentes do domínio)

  – variáveis

    denotadas por letras maiúsculas

- Isto é sempre possível, já que o conjunto **{0, 1, +, ·}** é completo.

Restrições Boleanas

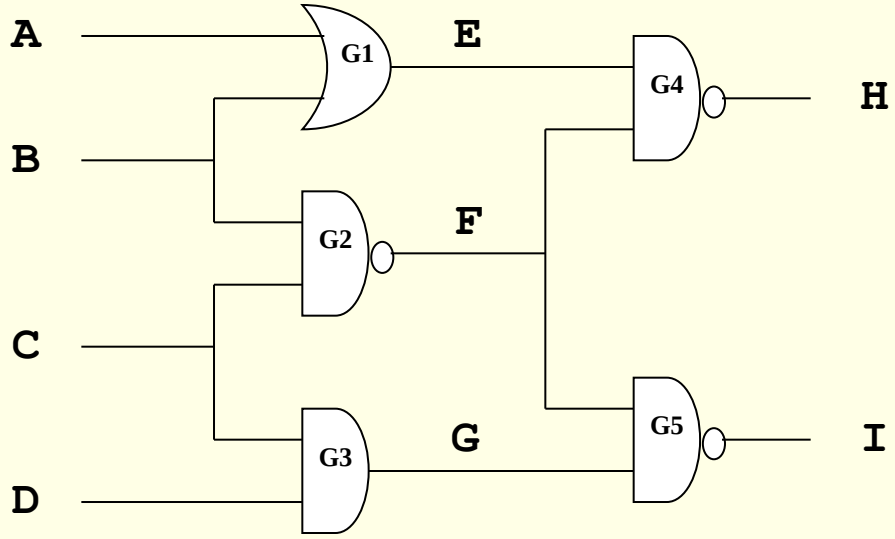
- Com efeito, dados os termos arbitrários *a* e *b*, todos os operadores e constantes podem ser expressos através destes operadores

$$a \wedge b = a \cdot b$$
$$a \vee b = a + b + a \cdot b$$
$$\neg a = 1 + a$$
$$a \rightarrow b = 1 + a + a \cdot b$$
$$a \leftrightarrow b = 1 + a + b$$

- Termos arbitrários serão denotados por minúsculas

Restrições Booleanas

- Circuitos digitais
  - Constantes 0 e 1
  - O operador + corresponde ao XOR
  - O operador · corresponde ao AND



- **E = A + B +A·B**
- **F = 1 +B·C**
- **G = C·D**
- **H = 1 + E·F**
- **I = 1 + F·G**

Unificação Boleana

- A implementação de CLP(B) mantém as restrições numa forma resolvida, obtida através da **unificação boleana**.

- Uma restrição boleana tem a forma $t_1 = t_2$ (em que os dois termos boleanos, t_1 e t_2 são formados exclusivamente a partir dos operadores $+$ e $\cdot$.

- A restrição boleana $t_1 = t_2$ pode ser satisfeita sse existir um **unificador boleano** para os dois termos t_1 e t_2.

- Um unificador boleano é uma substituição de variáveis por termos boleanos que garante que os dois termos tomam o mesmo valor boleano.

- Os unificadores boleanos serão designados através de letras gregas.

Unificação Booleana

- Exemplo: Os termos $t_1 = 1 + A$ e $t_2 = A \cdot B$ podem ser unificados com o unificador

$$\beta = \{A/1,\ B/0\}$$

- Com efeito, denotando por $t\uparrow\beta$ (ou simplesmente $t\beta$) a aplicação da substituição β ao termo t, temos

$$t_1\uparrow\beta = (1+A)\uparrow\{A/1,\ B/0\} = 1 + 1 = 0$$
$$t_2\uparrow\beta = (A \cdot B)\uparrow\{A/1,\ B/0\} = 1 \cdot 0 = 0$$

o que garante a satisfação da restrição de igualdade $t_1 = t_2$.

Unificação Booleana

- Em geral, dados dois termos Booleanos, t_1 e t_2, pode haver mais do que um unificador mais geral.

- Exemplo: A unificação dos termos $t_1 = 1 + A \cdot B$ e $t_2 = C + D$ pode ser obtida por qualquer um dos seguintes unificadores mais gerais

$$\beta_1 = \{ C / 1 + A \cdot B + D \}$$
$$\beta_2 = \{ D / 1 + A \cdot B + C \}$$

- Com efeito,

$t_1 \uparrow \beta_1 = (1+A \cdot B) \uparrow \{C/1+A \cdot B+D\} = 1+A \cdot B$

$t_2 \uparrow \beta_1 = (C+D) \uparrow \{C/1+A \cdot B+D\} = (1+A \cdot B+D)+D = 1+A \cdot B$

e

$t_1 \uparrow \beta_2 = (1+A \cdot B) \uparrow \{D/1+A \cdot B+C\} = 1+A \cdot B$

$t_2 \uparrow \beta_2 = (C+D) \uparrow \{D/1+A \cdot B+C\} = C+(1+A \cdot B+C) = 1+A \cdot B$

Unificação Booleana

- Existem outros unificadores (menos gerais) que podem ser obtidos através de instâncias dos anteriores, isto é, da composição de unificadores mais gerais com outras substituições.

- Por exemplo, a substituição λ obtida pela composição de β_1 com a substituição $\{A/0\}$

$$\lambda = \{C/1+A\cdot B+D\} \circ \{A/0\}$$
$$= \{C/1+D, A/0\}$$

ainda é um unificador dos termos $t_1=1+A\cdot B$ e $t_2=C+D$

$$t_1 \uparrow \lambda = (1+A\cdot B) \uparrow \{C/1+D, A/0\} = 1+0\cdot B = 1$$
$$t_2 \uparrow \lambda = (C+D) \uparrow \{C/1+D, A/0\} = (1+D+D) = 1$$

Restrições Boleanas

- Desta forma a restrição **X+X·Z = Y·Z+1** é satisfazível, já que a unificação dos termos **X+X·Z** e **Y·Z+1** sucede retornando o unificador boleano mais geral

$$\beta = \{X/Z \cdot\_X + Z + 1,\ Y/(1+Z) \cdot\_Y + Z\}$$

- Podemos confirmar este resultado, verificando que

$$\begin{aligned} (X+X \cdot Z)_{\uparrow \beta} &= (Z \cdot\_X + Z + 1) + (Z \cdot\_X + Z + 1) \cdot Z \\ &= (Z \cdot\_X + Z + 1) + Z \cdot\_X \\ &= Z + 1 \end{aligned}$$

e que

$$\begin{aligned} (Y \cdot Z+1)_{\uparrow \beta} &= ((1+Z) \cdot\_Y + Z) \cdot Z + 1 \\ &= Z + 1 \end{aligned}$$

Restrições Booleanas

$$\beta = \{X/Z \cdot\_X + Z + 1,\ Y/(1+Z) \cdot\_Y + Z\}$$

- Podemos pois concluir que a restrição **X + X·Z = Y·Z + 1** pode ser satisfeita independentemente do valor da variável **Z**, dado que o unificador mais geral β não a menciona)

- Numa análise mais pormenorizada
  - se **Z=0** então **X = 1** e **Y =_Y** (i.e. **Y** pode tomar qualquer valor)
  - se **Z=1** então **X =_X** e **Y = 1** (i.e. **X** pode tomar qualquer valor)

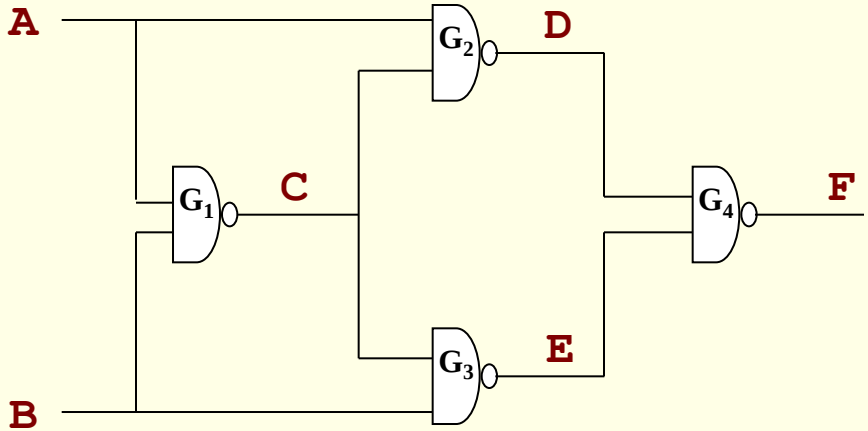
- O unificador β tem pois como instâncias fechadas (***ground***), os unificadores

{X/1, Y/0, Z/0},
{X/1, Y/1, Z/0},
{X/0, Y/1, Z/1},
{X/1, Y/1, Z/1},

Aplicações

- Um domínio de eleição para a utilização de restrições boleanas é o domínio dos circuitos digitais.

- Exemplo:
  1. Modelar o circuito abaixo através de um conjunto de restrições boleanas
  2. Verificar em que condições a saída toma o valor 1.



R_1: $C = 1 + A \cdot B$

R_2: $D = 1 + A \cdot C$

R_3: $E = 1 + B \cdot C$

R_4: $F = 1 + D \cdot E$

Restrições Boleanas no SICStus Prolog

Uma vez carregado no SICStus Prolog o módulo de Restrições boleanas, através da directiva

:- use_module(library(clpb)).

as restrições boleanas podem ser verificadas através do predicado **`sat(E)`**, em que a igualdade é representada por **`'=:='`** e os operadores **`+`** e **`·`** através de **`#`** e **`*`**, respetivamente.

Exemplos:

```
1.    ?- sat(A#B=:=F).
      sat(A=:=B#F)?
```

% A restrição A+B=F é satisfazível, com a substituição A/B+F

Restrições Boleanas no SICStus Prolog

2. `?- sat(A*B=:=1#C*D).`

   `sat(A=\=C*D#B)?`

% A restrição **A·B=1+C·D** é satisfazível, com substituição **A/1+C·D+B**

3. `?- sat(A#B=:=1#C*D), sat(C#D=:=B).`

   `sat(B=:=C#D),`

   `sat(A=\=C*D#C#D) ?`

% As restrições **A+B=1+C·D** e **B=C+D** são satisfeitas com a substituição **{A/1+C·D+C+D, B/C+D}**

Nota: Atenção à notação das respostas.

A=:= Exp	corresponde a	**A / Exp**
A=\= Exp	corresponde a	**A / 1+ Exp**

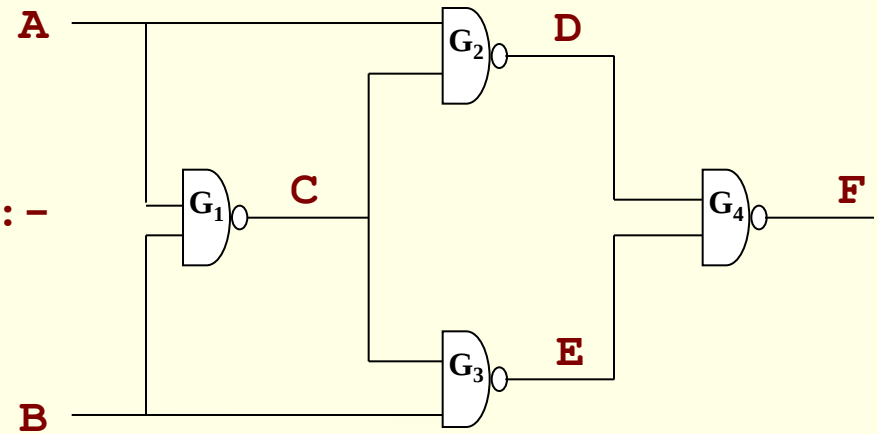
Restrições Boleanas no SICStus Prolog

Um exemplo mais completo, relativo ao circuito anterior

```
:- use_module(library(clpb)).

nand_gate(X,Y,Z):-
    sat(X*Y =:= 1#Z).

circuit(A,B,[C,D,E],F):-
    nand_gate(A,B,C),
    nand_gate(A,C,D),
    nand_gate(B,C,E),
    nand_gate(D,E,F).
```

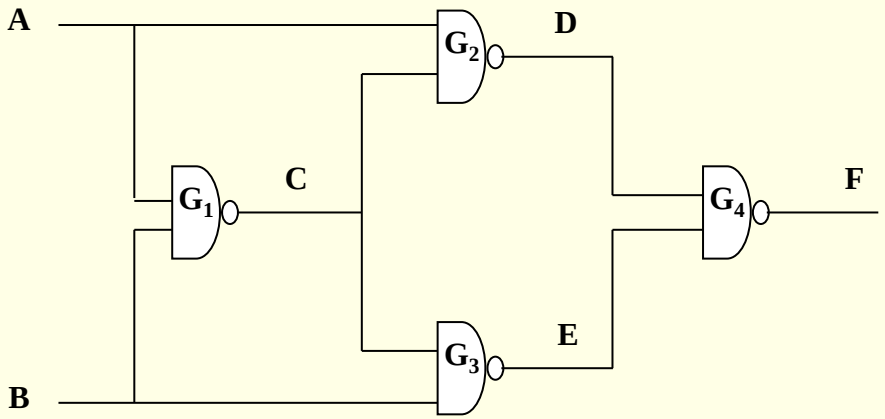


Restrições Boleanas no SICStus Prolog

Alguma interação com o sistema

```
| ?- circuit(A,B,[C,D,E],F).
     sat(C=:=_A*F#F#_A),
     sat(A=\=_A*F#_B*F#F#_A),
     sat(B=\=_A*F#_B*F#_A),
     sat(D=\=_B*F),
     sat(E=\=_B*F#F)

| ?- circuit(A,B,[C,D,E],1).
C = 1,
E = A,
sat(B=\=A),
sat(D=\=A)
```



CLP(B) no SICStus

```
:- use_module(library(clpb)).
```

```
sat(+Expression)
```
(com ~, *, +, #, =:=, <, …)

```
taut(+Expression, -Truth)
labeling(+Variables)
```

E.g.

```
?- sat( card([2-3],[X,Y,Z,X+W]) ).
```