Le langage LUSTRE : Programmation data-flow synchrone pour les systèmes embarqués

N7 3SN

Cours 3: vérification de programmes par model checking

Frédéric Boniol

ONERA - 2, av. E. Belin - 31055 Toulouse

frederic.boniol@onera.fr

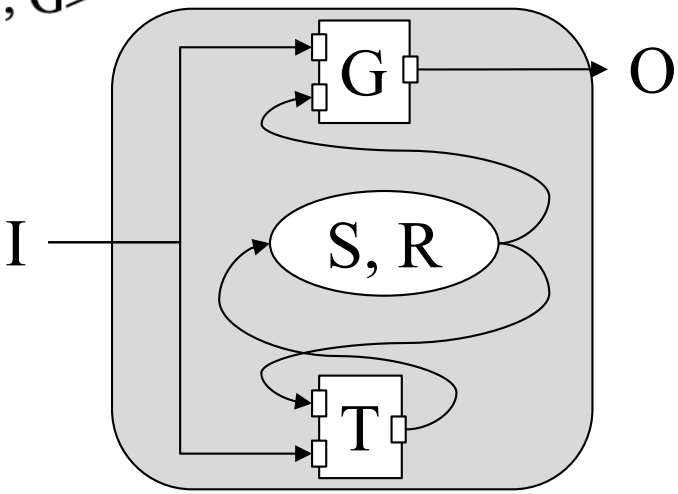
1. Principes (1/1)

Rappel : un programme Lustre décrit un comportement exprimable par un automate fini

⇒ Représentable par une machine Mealy déterministe

Machine de Mealy déterministe : M = <S , s_0, R, I, O T, G>

- S = ensemble des états de la machine
- s_0 = état initial (appartenant à S)
- R = registres internes de la machine (c'est sa mémoire)
- I = input de la machine
- O = output de la machine
- $T = S \times val(R) \rightarrow S \times val(R)$ fonction de transition totale
- $G = S \times val(R) \rightarrow O$ fonction de sortie totale



$O_t = G((S,R)_t , I_t)$

$(S, R)_{t+1} = T((S,R)_t , I_t)$

La sortie dépend de l'entrée, de l'état présent, et de la mémoire présents

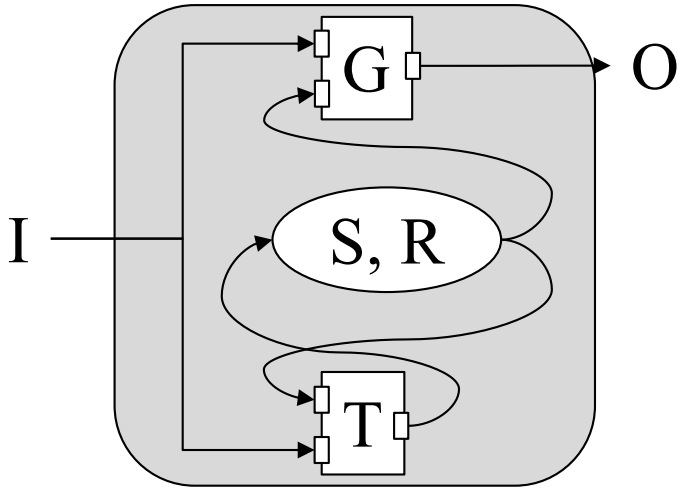
L'état futur dépend de l'entrée, de l'état présent, et de la mémoire présents

Avancement synchrone avec un horloge discrète

1. Principes (1/2)

Lustre versus machine de Mealy déterministe :

- Notion d'état = valeur des variables bool
- Notion de registres = valeur des flots int et float
- Notion de transition (effet d'une réaction à des entrées sur les variables d'état et les sorties du programme).



$O_t = G((S,R)_t , I_t)$

$(S, R)_{t+1} = T((S,R)_t , I_t)$

- Complétude / causalité des équations Lustre => G et T sont des fonctions totales (d'où le déterminisme)

1. Principes (1/2)

Example :

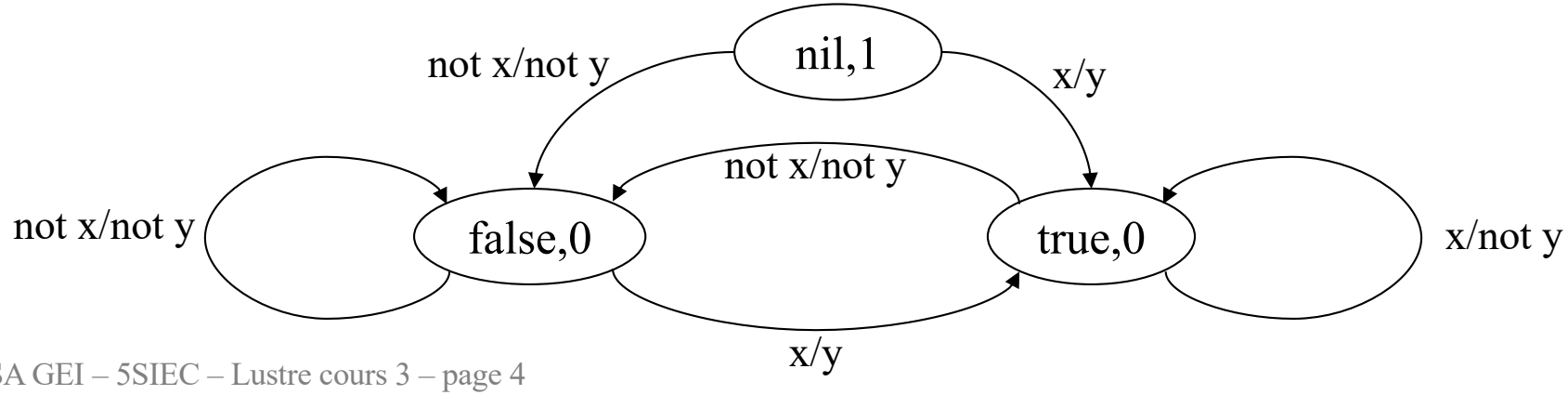
```
node rising_edge (X : bool) returns (Y : bool) ;
let
   Y = X -> (X and not pre(X));
tel ;
```

=> Fonction de réaction :

```
init = 1;
At each step :
        Y = if (init) X else (X && ! pX);
        pX = X
        init = 0
```

=> Etat = (pX, init), input = X, output = Y

=> Machine de Mealy sous-jacente



1. Principes (1/2)

Idée :

- En calculant les successeurs de chaque état atteint, on obtient l'arbre d'exécution infini du programme.
- En regroupant les états identiques (ayant les mêmes valeurs pour toutes les variables qui les composent), on obtient un graphe orienté représentant l'automate explicite du programme, déplié sur toutes les variables internes du programme (la machine de Mealy).
- On peut analyser les comportements du programme en étudiant la structure de ce graphe :
  - les états qui le composent (et les valeurs de certains signaux dans ces états),
  - les successions d'états le long d'un chemin,
  - les composantes fortement connexes, ...

  $\Rightarrow$ On peut « vérifier » que le programme satisfait des propriétés données

Question : quelles propriétés et comment les exprimer ?

1. Principes (2/2)

Idée :

- Lustre peut être vu comme une logique temporelle du passé permettant la spécification de comportements attendus… puis leur vérification

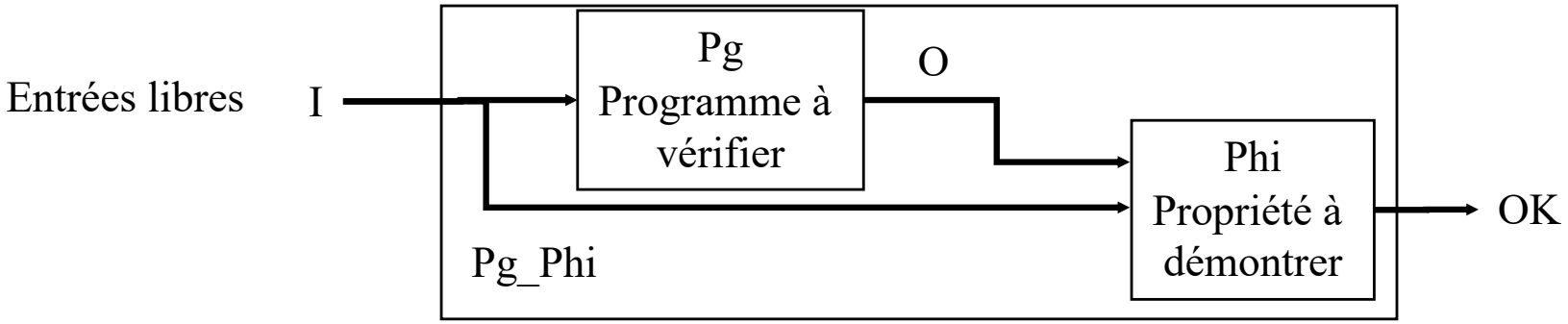
Mais

- Seuls les comportements attendus équivalents à des propriétés de sûreté peuvent être vérifiés (« une situation indésirable ne peut **jamais** se produire »)

Principe :

- expression des propriétés attendues sous la forme d'un nœud LUSTRE retournant un flot booléen OK (ce flot booléen dénote la valeur de vérité de la propriété attendue)
- composition de ce nœud avec le programme à vérifier et internalisation de toutes les sorties sauf OK
- L'ensemble $O = \{OK\}$ est l'ensemble des sorties de la machine de Mealy correspondante
- Parcours de la machine de Mealy
  - S'il existe une séquence de transitions telle que OK=fasle, alors la propriété est déclarée fausse
  - Sinon, la propriété est déclarée vraie

2. Architecture de vérification



- Pg = programme à vérifier écrit en LUSTRE
- Phi = observateur (propriété à démontrer écrite en LUSTRE)

⇒ Pg_Phi = Pg || Phi = composition synchrone du programme et la propriété

⇒ Pg_Phi est un programme écrit en LUSTRE !

⇒ Phi est satisfaite ssi tous les états de Pg_Phi sont étiquetés par la valeur true sur le flot OK

⇒ Vérification de Pg = composition synchrone des machines de Mealy de Pg_Phi et de Phi et test des valeurs prises par OK dans chaque état
= model checking !

⇒ Outils : LESAR et NP-tools (intégrés dans SCADE)
permet la vérification de programmes mettant en œuvre des flots booléens et des flots numériques définis par des relations linéaires

2. Architecture de vérification

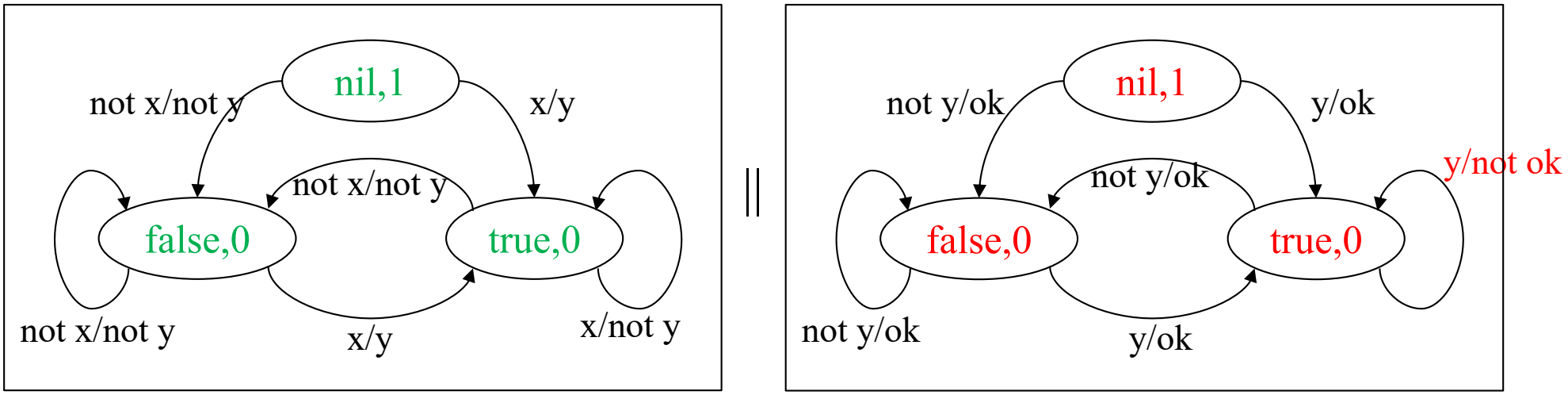
Example :

```
node rising_edge (X : bool) returns (Y : bool) ;
let
   Y = X -> (X and not pre(X));
tel ;
```

=> Propriété : Y n'est jamais vrai deux instants de suite

```
node phi (Y : bool) returns (OK : bool) ;
let
   OK = true -> (not (Y and pre(Y));
tel ;
```

=> Machines de Mealy sous-jacente



rising_edge phi

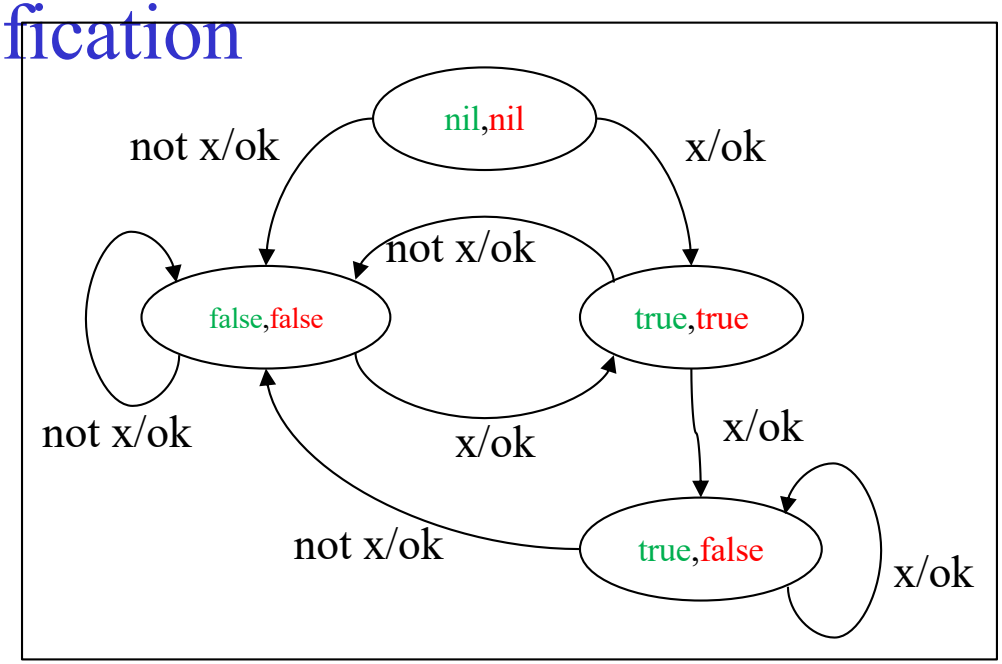
2. Architecture de vérification

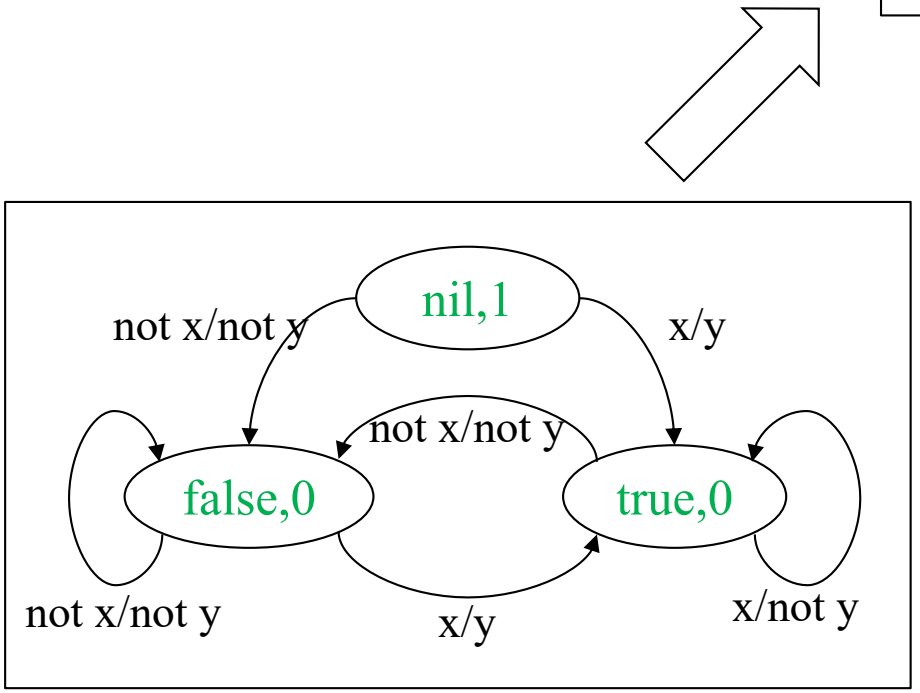
Example :

=> Composition synchrone

=> Ok est toujours vrai

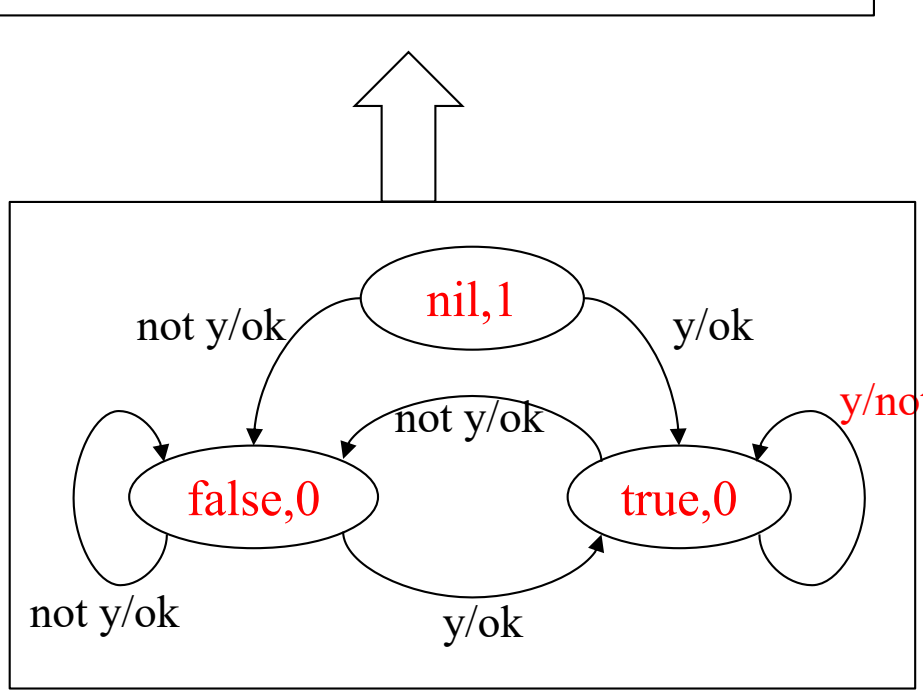
=> La propriété est vraie !





rising_edge

||



phi

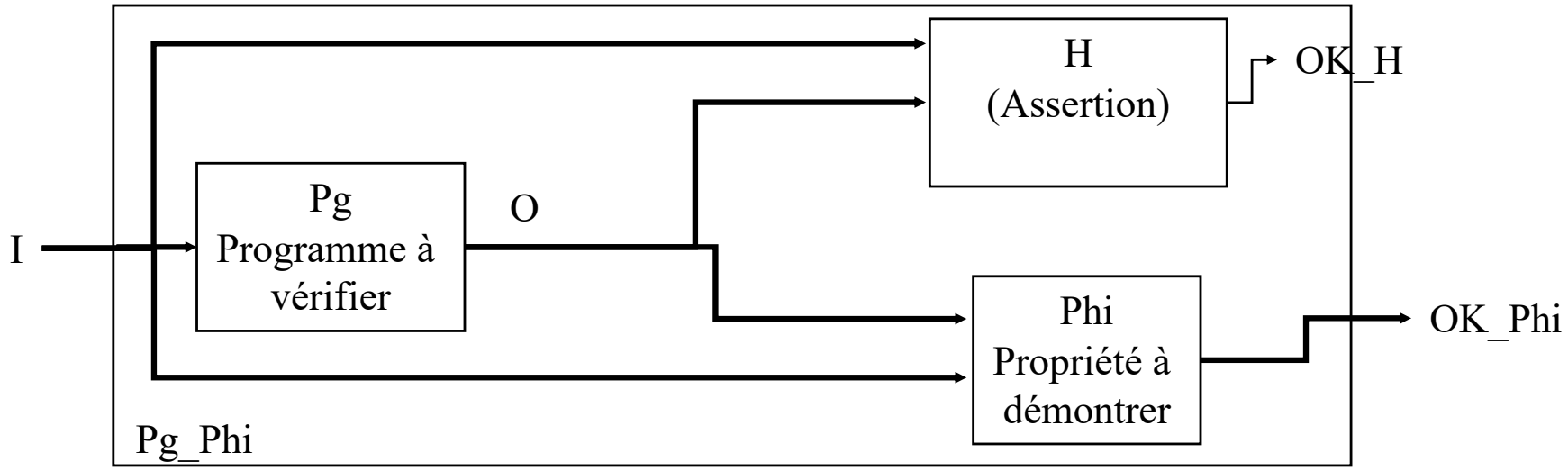
2. Architecture de vérification + assertions

Idée : Des assertions permettent de tenir compte de l'environnement du programme en exprimant des contraintes sur ses entrées en fonction de ses sorties

Une assertion est une formule logique sur les flots d'entrée et de sortie, qui doit être toujours vraie

⇒ Permet de vérifier que Pg satisfait Phi sous l'hypothèse que ses entrées et ses sorties satisfont l'assertion H

⇒ Permet de restreindre l'espace d'exploration



3. Expression des propriétés en Lustre

Observateur :

programme qui calcule un (et un seul) flot booléen en fonction de ses entrées (de type quelconque)

→ permet de modéliser des propriétés fonctionnelles attendues

Exemple :

```
node never (A : bool) returns (B : bool);
let
   B = not A -> (not A) and pre(B);
tel.
```

Never(X) est vrai tant que X est faux.

3. Expression des propriétés en Lustre

Exemple :

Soit la propriété attendue suivante

« toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C »

Peut être exprimée au passé par

« à chaque occurrence de C, il faut que A ne se soit jamais produit, ou si A s'est produit, il faut que B se soit produit depuis la dernière occurrence de A »

Exprimable en LUSTRE par le nœud

```
node once_B_from_A_to_C (A, B, C: bool)returns (X: bool);
let
  X = C => (never(A) or since(B,A));
tel.

node since (X, Y: bool) returns (Z: bool)
let
  Z = if Y then X else (true -> X or pre(Z));
tel.
```