Soutenance de stage de recherche (UC Berkeley)

Certificate-Based Combinational Equivalence Checking

20/08/2025

Rémi Germe

1 Contexte

Contexte

Industrie des semi-conducteurs & Electronic Design Automation (EDA)

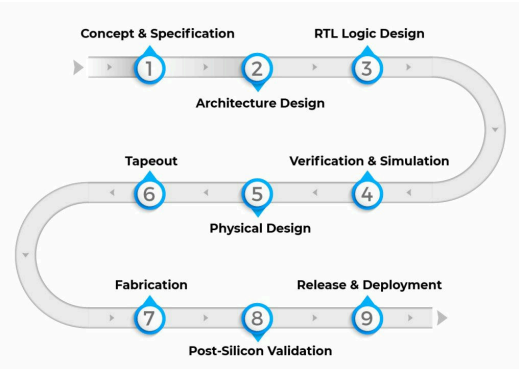


Image issue du blog de vvdntech.com

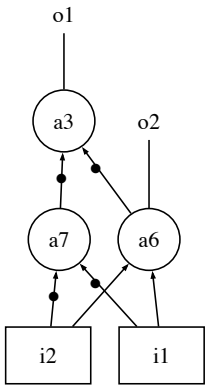
- Nombreuses représentations différentes
- Besoin de s'assurer que chacune des étapes préserve la sémantique/le comportement du circuit
- Focus sur l'équivalence fonctionnelle de circuits combinatoires

- Période initiale ($\sim$ 1 mois) : recherche d'un sujet **pertinent** et **réalisable** : difficulté d'évaluation de ces critères car manque de recul.

2 État de l’art

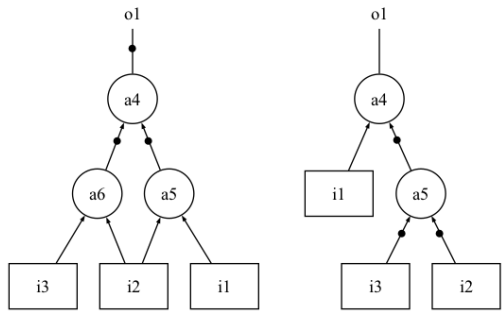
And-Inverter Graphs (AIGs) & optimisation logique

AIG



fanin: entrée (fanins de $a_6 = \{i_1, i_2\}$)
fanout: sortie (fanouts de $a_6 = \{o_2, a_3\}$)

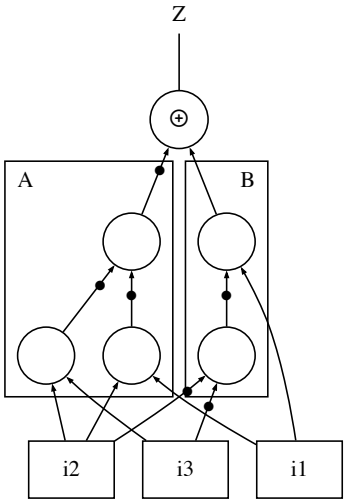
Passes d'ABC utilisées dans Yosys : `rewrite`, `refactor`, `balance`. Exemple de `rewrite` : $(i_1 \wedge i_2) \vee (i_1 \wedge i_3)$ et $i_1 \wedge (i_2 \vee i_3)$:

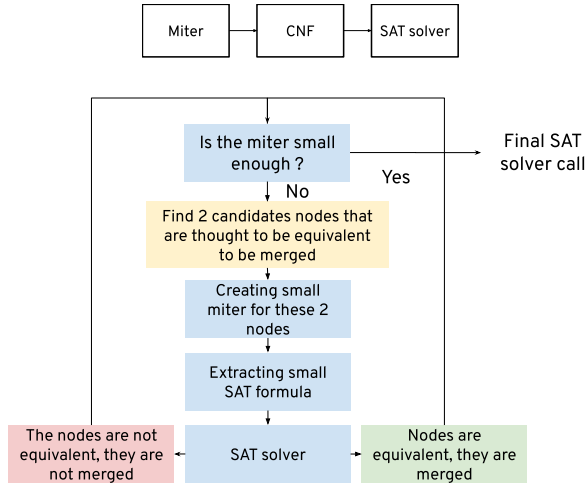


Le "meilleur" dépend du reste du circuit !

Combinational Equivalence Checking (CEC)

Un miter entre $(i_1 \wedge i_2) \vee (i_1 \wedge i_3)$ et $i_1 \wedge (i_2 \vee i_3)$



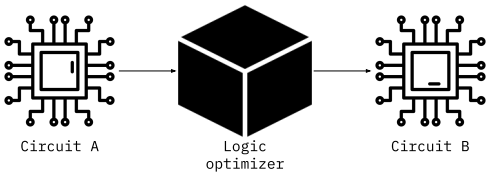


Techniques existantes : simulation pour identifier des classes d'équivalence (potentielle) de noeuds.

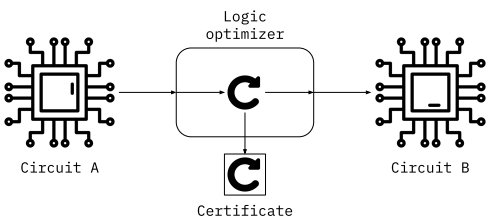
3 Certificate-Based CEC

Motivations

Actuellement :



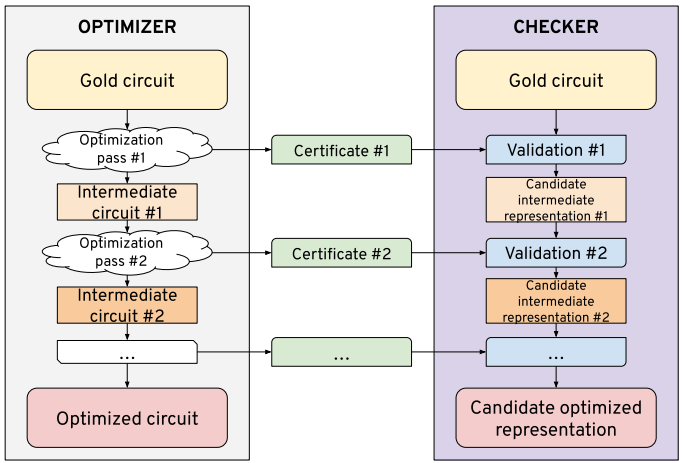
Notre approche :



intuition : valoriser l'information connue lors de l'optimisation

Format des certificats

Les certificats ne sont pas supposés corrects (car émis par l'optimizer à vérifier).



Pouvoir suivre les modifications faites par l'optimizer
⇒ **mutations** `create` et `replace`

Faciliter la preuve d'équivalence en pointant des noeuds censés être équivalents
⇒ **indices**

Utilisation des certificats

Algorithm 1: Naive Equivalence

```
1:  procedure NAIVE-EQ(C_i, C_{i+1}, hints)
2:      M ← create-miter(C_i, C_{i+1})
3:      for n ∈ C_i in topological order do
4:          if n ∈ hints then
5:              cnf ← extract-cnf(M, n, hints[n])
6:              if is-unsat(cnf) then
7:                  merge(M, n, hints[n])
8:              end
9:          else if n is obviously mergeable then
10:             merge(M, n, n)
11:         end
12:     end
13:     return miter-outputs-merged(M)
14: end
```

Obviously mergeable: même fanins (déjà merged au préalable)

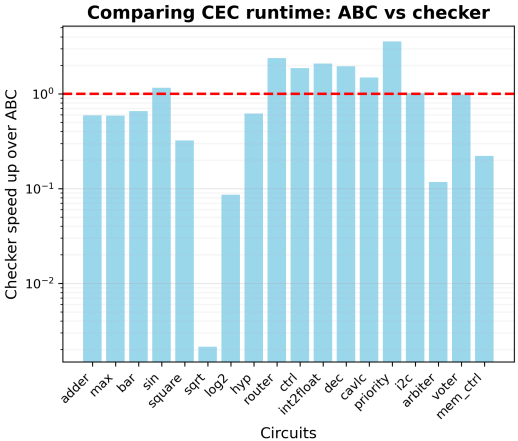
Problème : trop de requêtes SAT (faciles) - va motiver une approche améliorée.

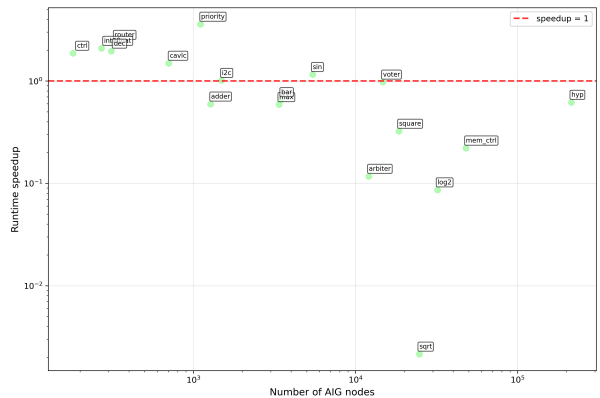
4 Premiers résultats

Premiers résultats

Setup : EPFL combinational benchmark, comparaison avant/après `rewrite`, 20 runs lancées simultanément

Forte variabilité des résultats (petits circuits)





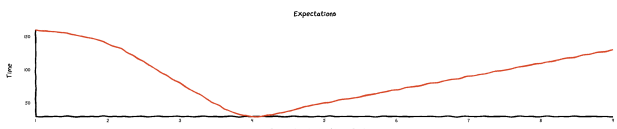
5 Travail futur

Travail futur

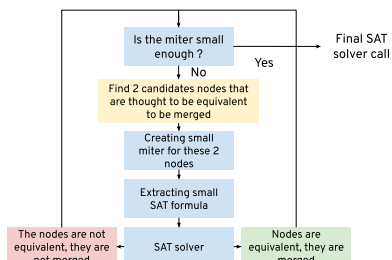
Émission des certificats

- ☑ `rewrite`
- ☑ `refactor`
- ☐ `balance` : structure différente des autres passes
- ◧ sérialisation : résolue en théorie, debug nécessaire

Stratégie de preuve améliorée : utilisation intelligente des certificats



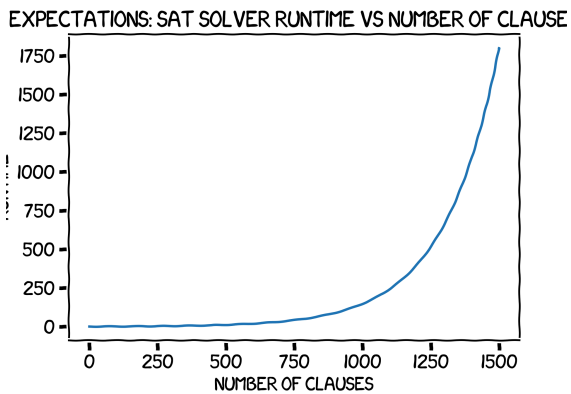
Trouver une heuristique pour évaluer le coût des requêtes SAT faites à partir d'un noeud. Tant que la requête est trop grosse, merge "les meilleurs" noeuds dans son fanin

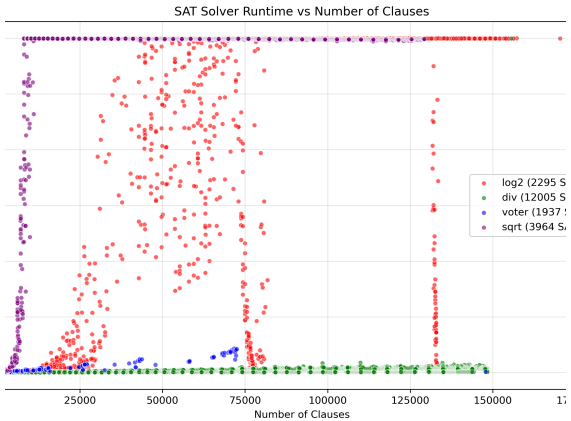


Travail futur (ii)

Coût des requêtes SAT

Attentes vs **Réalité**

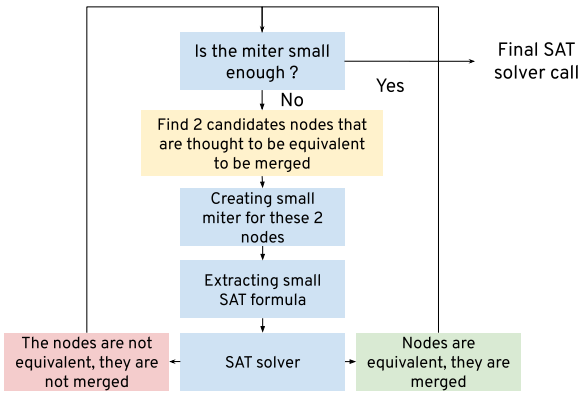


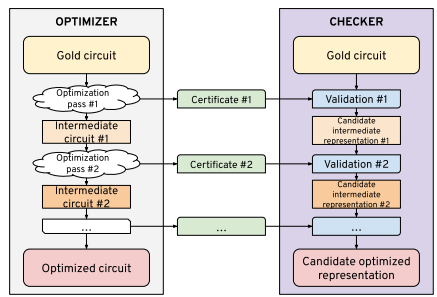


À faire : heuristique plus fine, regarder les travaux existants (notamment circuit-aware SAT solvers).

Conclusion

- Design et implémentation d'une nouvelle approche
- Premiers résultats relativement encourageants mais sérieuses difficultés soulevées
- Pistes pour améliorer la stratégie de preuve





Merci pour votre attention.

6 Annexe

Rewrite

Essaie de remplacer des sous-graphes par une représentation équivalente pré-calculée.

NPN-equivalence : Negation-Permutation-Negation

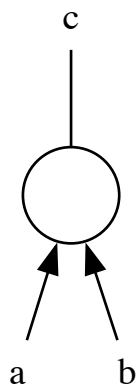
- Seulement 222 classes de NPN-équivalence pour les fonctions à 4 entrées ($\ll 2^{16}$).
- Pour chaque classe, différentes représentations sont pré-calculées.
- Algo naïf : on essaie toutes les représentations et on évalue leur coût (nombre de noeuds rajoutés / supprimés), on garde la meilleure

---

- Pour chaque noeud, on calcule différentes 4-coupes
- On fait ça pour tous les noeuds dans un ordre topologique

Transformation de Tseitin

Permet de générer des clauses SAT traduisant des contraintes booléennes.



a	b	$c = a \wedge b$	SAT clause
0	0	0	$a \vee b \vee \neg c$
0	1	0	$a \vee \neg b \vee \neg c$
1	0	0	$\neg a \vee b \vee \neg c$
1	1	1	$\neg a \vee \neg b \vee c$

La formule SAT générée pour une porte `AND` est :

$$(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$$
$$\wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$$

ce qui se simplifie en

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c).$$