Sensibilisation à la vérification dans la conception des systèmes numériques

Bertrand LE GAL
bertrand.legal@ims-bordeaux.fr

Laboratoire IMS - UMR CNRS 5218
Institut Polytechnique de Bordeaux
Université de Bordeaux
France

Exemples introductifs

Exemples introductifs à la problématique...

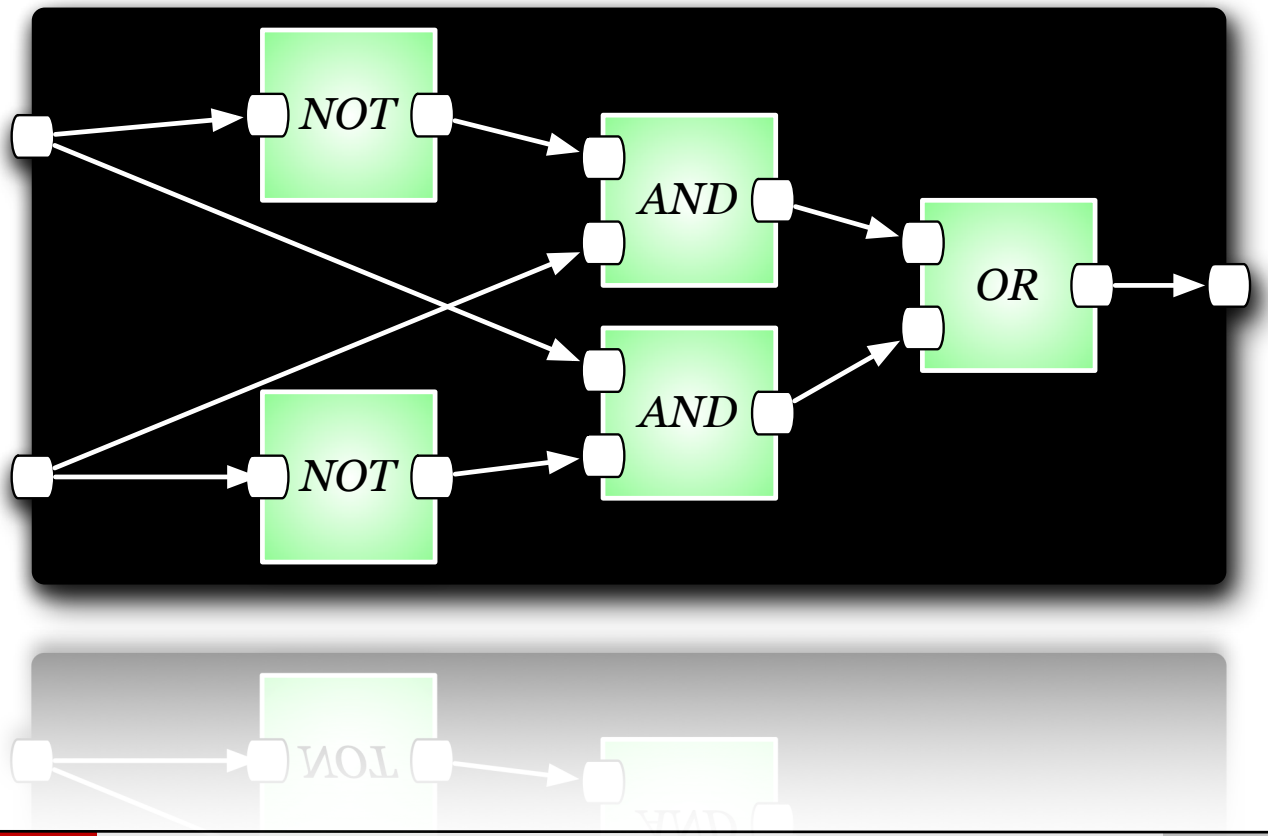
```
void ConversionCouleurs(unsigned char*r, unsigned char*v, unsigned char*b,
                        unsigned char*y, unsigned char*cb, unsigned char*cr){
     short sy  = round(        0.299   * (double)(*r) + 0.587   * (double)(*v) + 0.114   * (double)(*b));
     short scb = round(128.0 - 0.16874 * (double)(*r) - 0.33126 * (double)(*v) + 0.5     * (double)(*b));
     short scr = round(128.0 + 0.5     * (double)(*r) - 0.41869 * (double)(*v) - 0.08131 * (double)(*b));
     *y  = (sy<0 )?0:(sy>255 )?255:sy;
     *cb = (scb<0)?0:(scb>255)?255:scb;
     *cr = (scr<0)?0:(scr>255)?255:scr;
}
```

Exemples introductifs à la problématique...

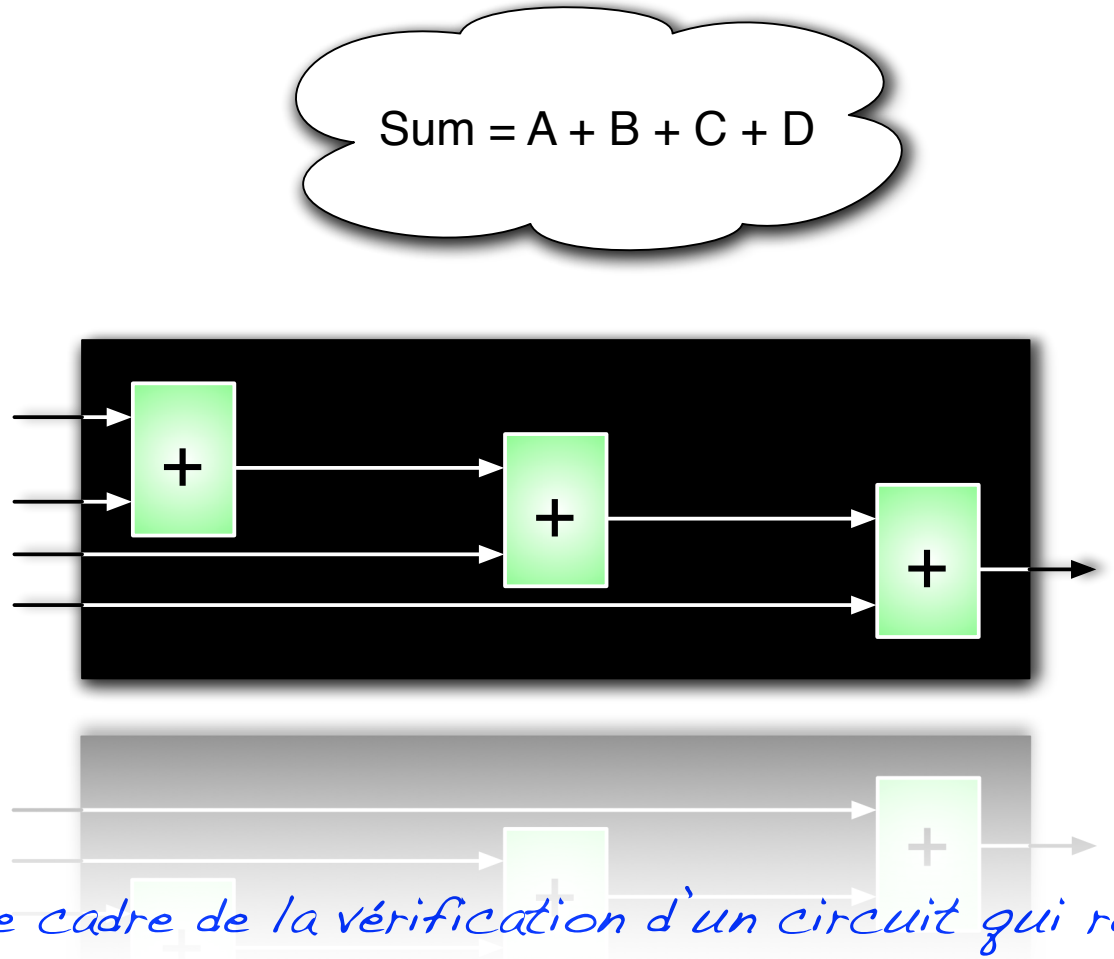
```
int RLE(short *in, short* out){
      int last     = in[0];
      int cpt      = 1;
      int ecriture = 0;

      for(int i=1; i<64; i++){
            if( (in[i] == last) && (cpt < 15) ){
                  cpt += 1;
            }else if(cpt != -1){
                  out[ecriture++] = ((cpt & 0x0F) << 12) + (last+2048);
                  last            = in[i];
                  cpt             = 1;
            }
      }
      out[ecriture++] = ((cpt & 0x0F) << 12) + (last+2048);
      return ecriture;
}
```

Exemple des portes logiques

Dans le cadre de la vérification d'une porte XOR et des portes élémentaires la composant, combien de cas différents doit on valider pour assurer un f(x) correct ?



Exemple de la somme de 4 données numériques



Dans le cadre de la vérification d'un circuit qui réalise la somme de 4 valeurs entières codées sur 16 bits,

=> Combien de validations réaliseriez vous ? Pourquoi ?

=> Quelles valeurs choisiriez vous ?

Exemple de multiplication par additions

- Nous souhaitons réaliser une multiplication à l'aide d'un algorithme itératif utilisant un additionneur (contrainte du système),
  - La ressource de calcul que nous allons réaliser doit permettre de réaliser des multiplications entre 2 nombres entiers codés sur 8 bits,
  - L'algorithme de multiplication itératif aura pour caractéristique de proposer des temps (nombre de cycles) de multiplication variant en fonction des arguments de calcul,

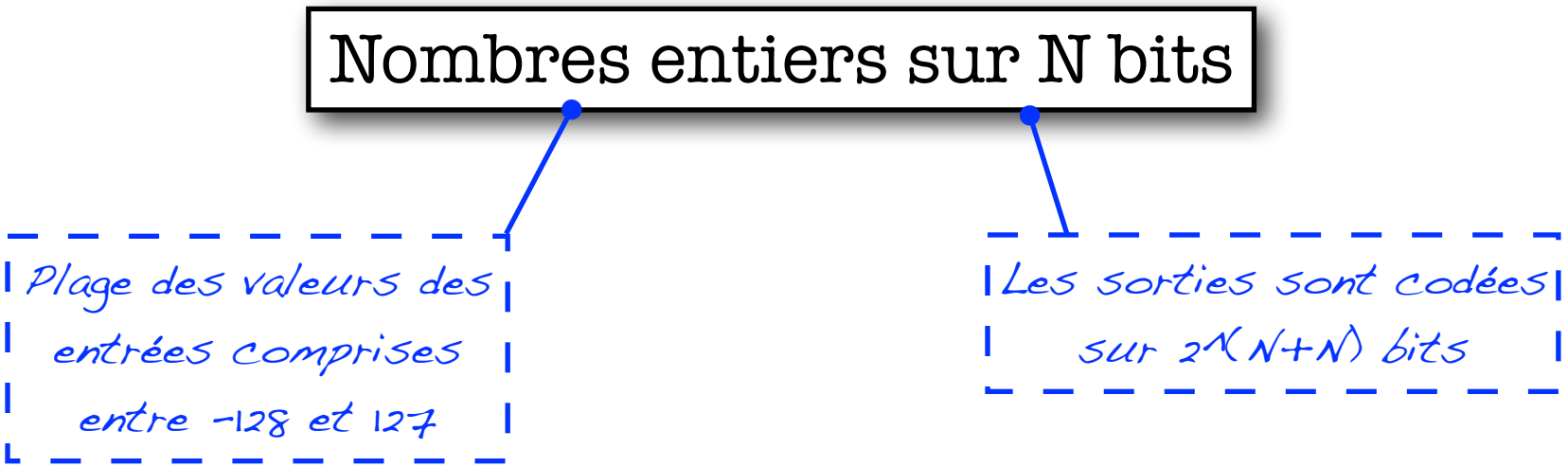
Exemple de réalisation d'une multiplication à base d'un additionneur utilisé itérativement

Ce composant est il cohérent vis-a-vis des spécifications ?

```
int mult(int a, int b){
    int c = 0;
    while( b != 0 ){
        c = c + a;
        b = b - 1;
    }
    return c;
}
```

Exemple de multiplication par additions

- Maintenant oubliez ce que vous avez vu sur la planche précédente et reprenez uniquement les hypothèses de fonctionnement afin de définir la procédure de validation de la fonction.
  - La fonction de calcul que nous devons réaliser doit permettre la multiplication de 2 nombres entiers codés sur 8 bits,
  - L'algorithme de multiplication est itératif. Il sera caractérisé par des temps (nombre de cycles) de multiplication variables en fonction des arguments.



Exemple de multiplication par additions

- Conditions de fonctionnement
  - La fonction de calcul que nous devons réaliser doit permettre la multiplication de 2 nombres entiers codés sur 8 bits,
  - L'algorithme de multiplication est itératif. Il sera caractérisé par des temps (nombre de cycles) de multiplication variables en fonction des arguments.
- Déterminez un jeu de valeurs permettant de valider intelligemment la fonction de calcul (le moins de valeurs possibles) :

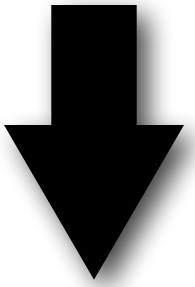
a								
b								
c								

Exemple de multiplication par addition

Exemple de réalisation d'une multiplication à base d'un additionneur utilisé itérativement

Ce composant est il cohérent vis-a-vis des spécifications ?

```
int mult(int a, int b){
    int c = 0;
    while( b != 0 ){
        c = c + a;
        b = b - 1;
    }
    return c;
}
```



Fonctionnement erroné dans le cas ou (b < 0)

Exemple de multiplication par addition

Cette version du composant est valide vis-a-vis des spécifications et implémente les quelques tests représentatifs des différents cas possibles :

$a = 2 \;\&\; b = 3 \Rightarrow c = 6$

$a = -2 \;\&\; b = 3 \Rightarrow c = -6$

$a = 2 \;\&\; b = -3 \Rightarrow c = -6$

$a = 2 \;\&\; b = 0 \Rightarrow c = 0$

$a = 0 \;\&\; b = 3 \Rightarrow c = 0$

$a = 0 \;\&\; b = 0 \Rightarrow c = 0$

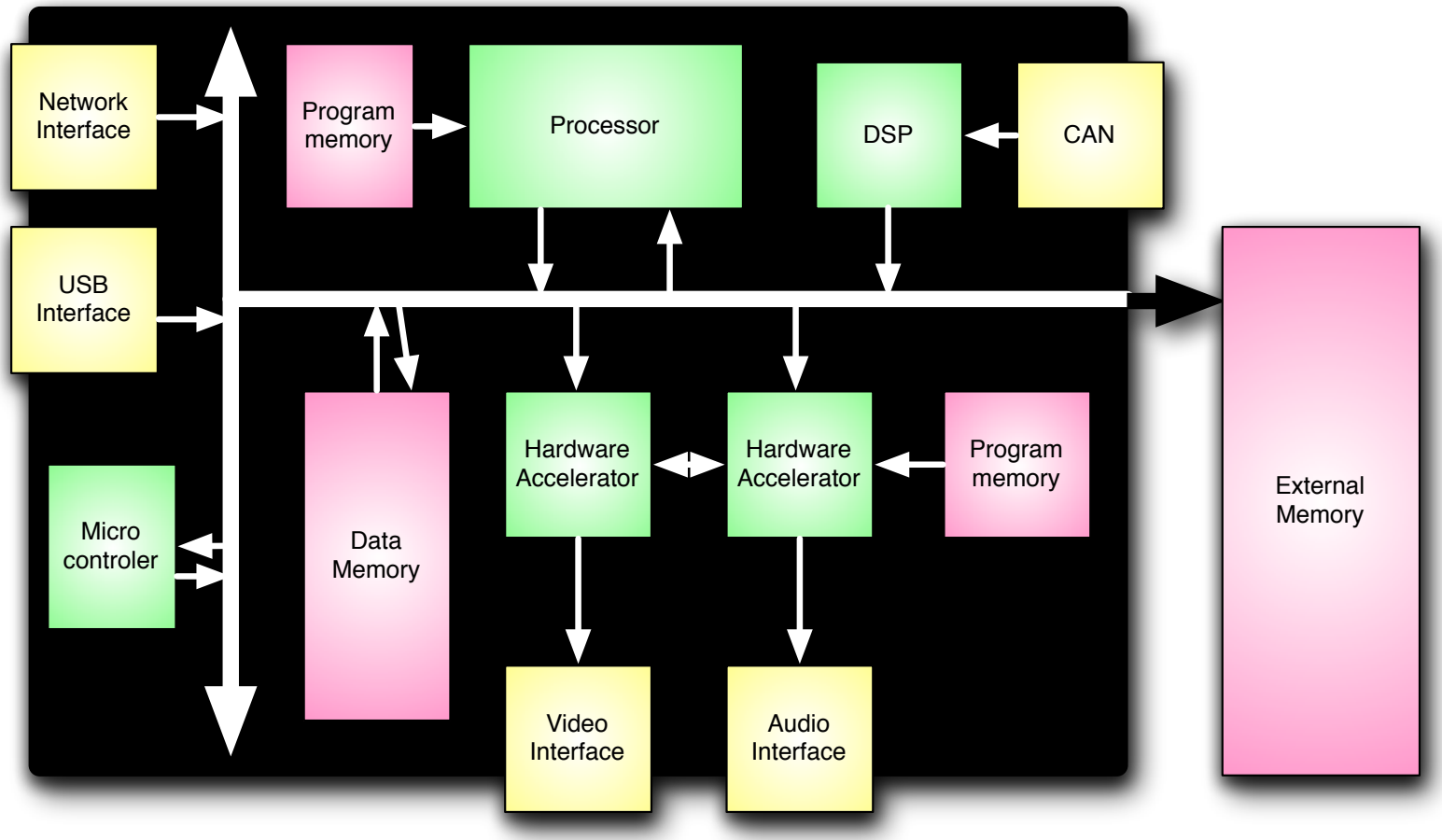
Rectifiez le comportement exprimé sous sa forme algorithmique en fonction de l'erreur que vous venez de détecter...

```
int mult(int a, int b){
   int c = 0;
   bool neg = (b>=0)?false:true;
   b = (b>=0)?b:-b;
   while( b != 0 ){
       c = c + a;
       b = b - 1;
   }
   return (neg)?c:-c;
}
```

Exemples de circuits issus du monde réel

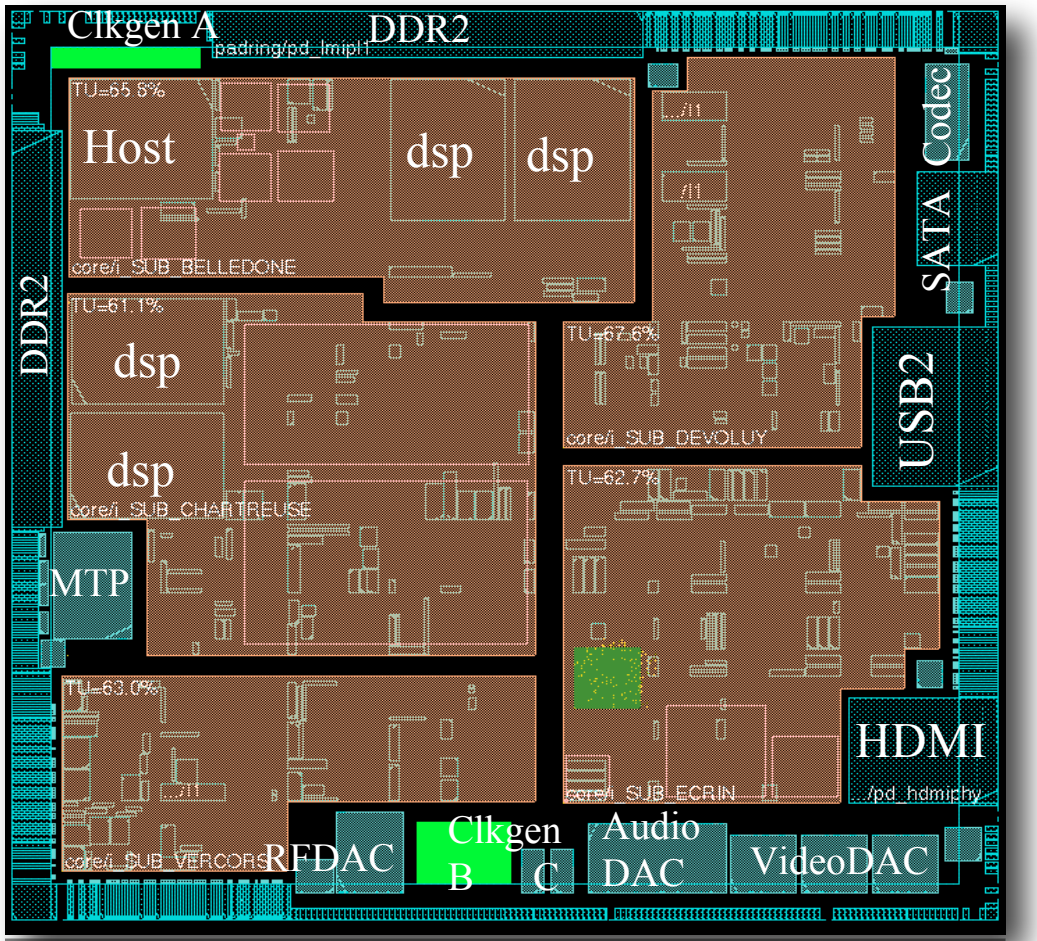
Qu'est ce qu'un système numérique aujourd'hui ?

Un système est un savant mélange de matériel et de logiciel assurant des traitements ainsi que du contrôle.



Exemple, le décodeur “HD 264” de STMicro

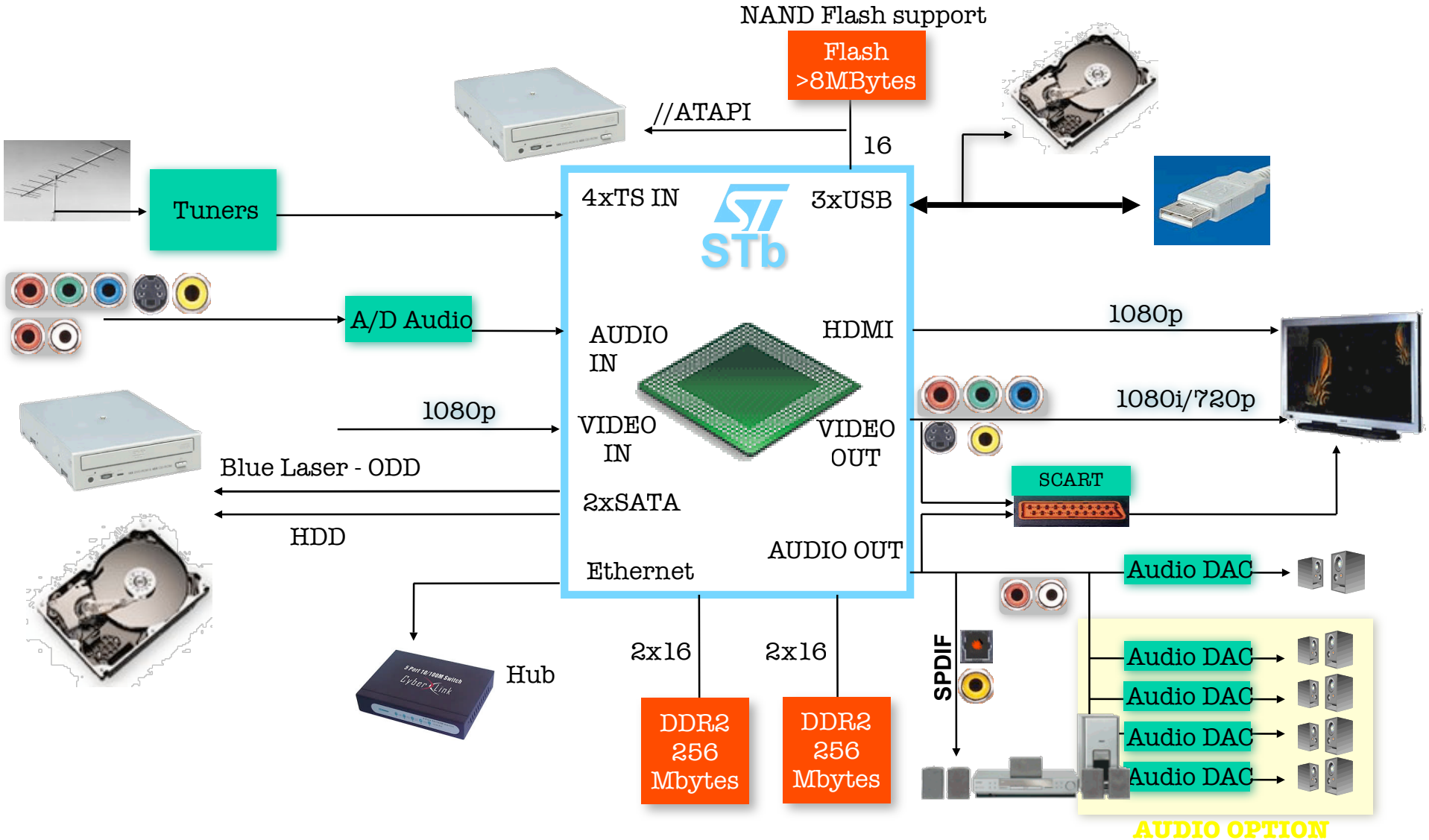
- Circuit dédié au décodage de la TV HD (norme H264)
  - Circuit contenant 150M transistors et 886 pads IOs (~5 GMIPS)
  - 128 sources d’intérruption
  - 73 initiateurs et 96 cibles sur les bus,
  - 115 réseaux d’horologe (19 pour les interconnexions),
- Construction du système
  - 4 processeurs (2 DSP pour la vidéo, 1 DSP pour l’audio et 1 généraliste pour la configuration),
  - 36 Soft IPs + 2 Hard IPs
  - 140 memory cuts



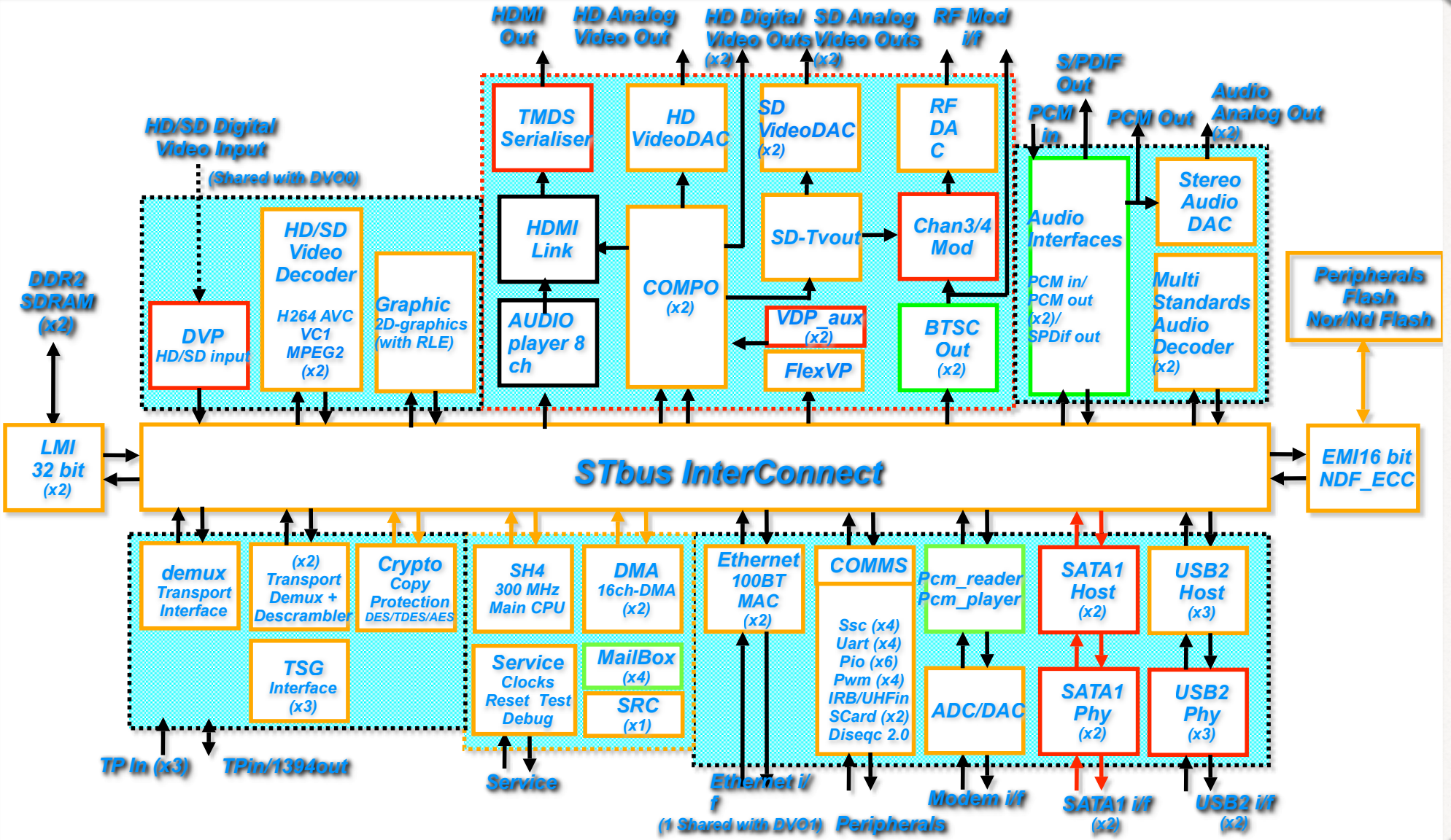
Circuit développé en 200X ?!

Que fait on alors aujourd'hui ...

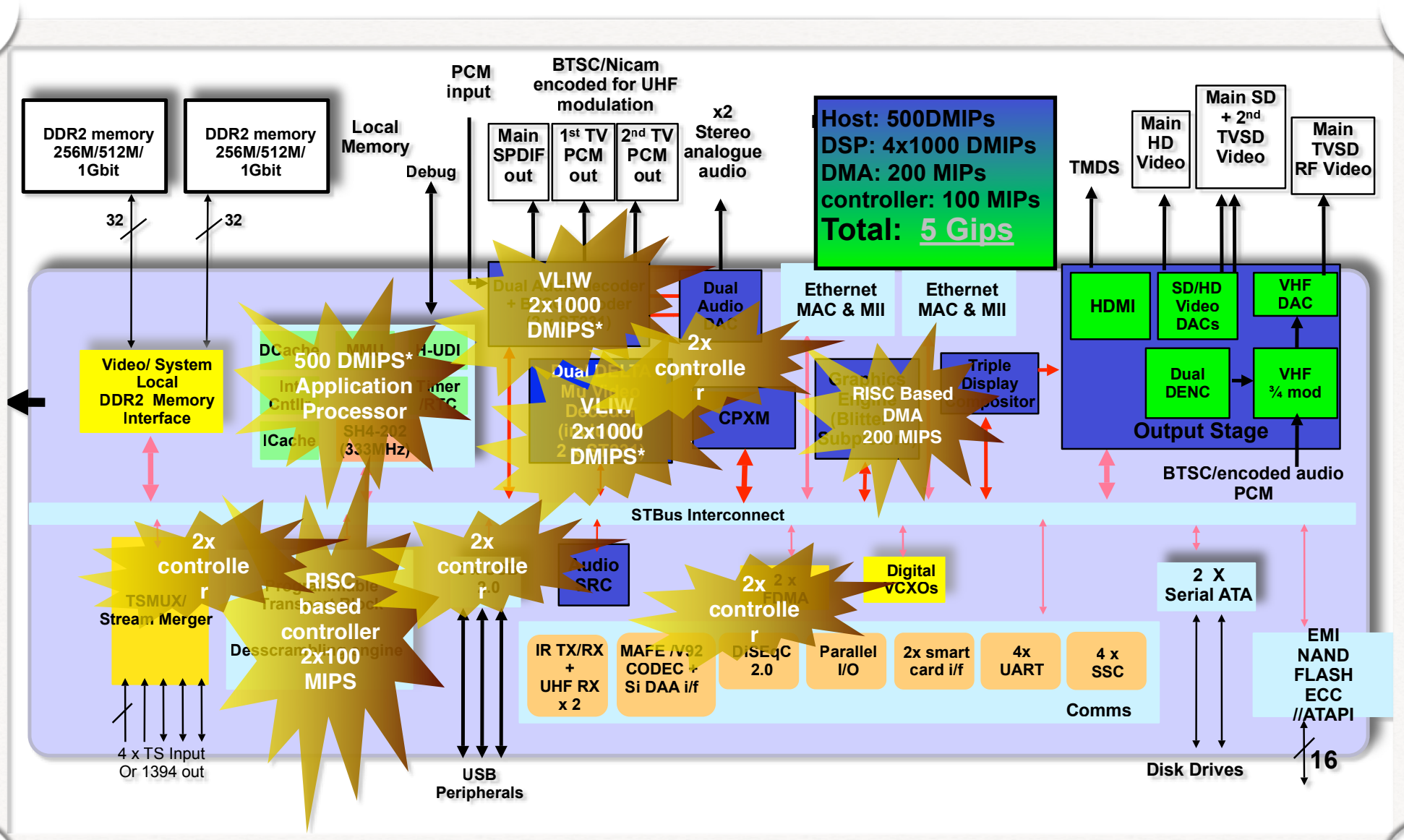
Exemple, le décodeur “H-264” de STMicro



Exemple, le décodeur “H-264” de STMicro

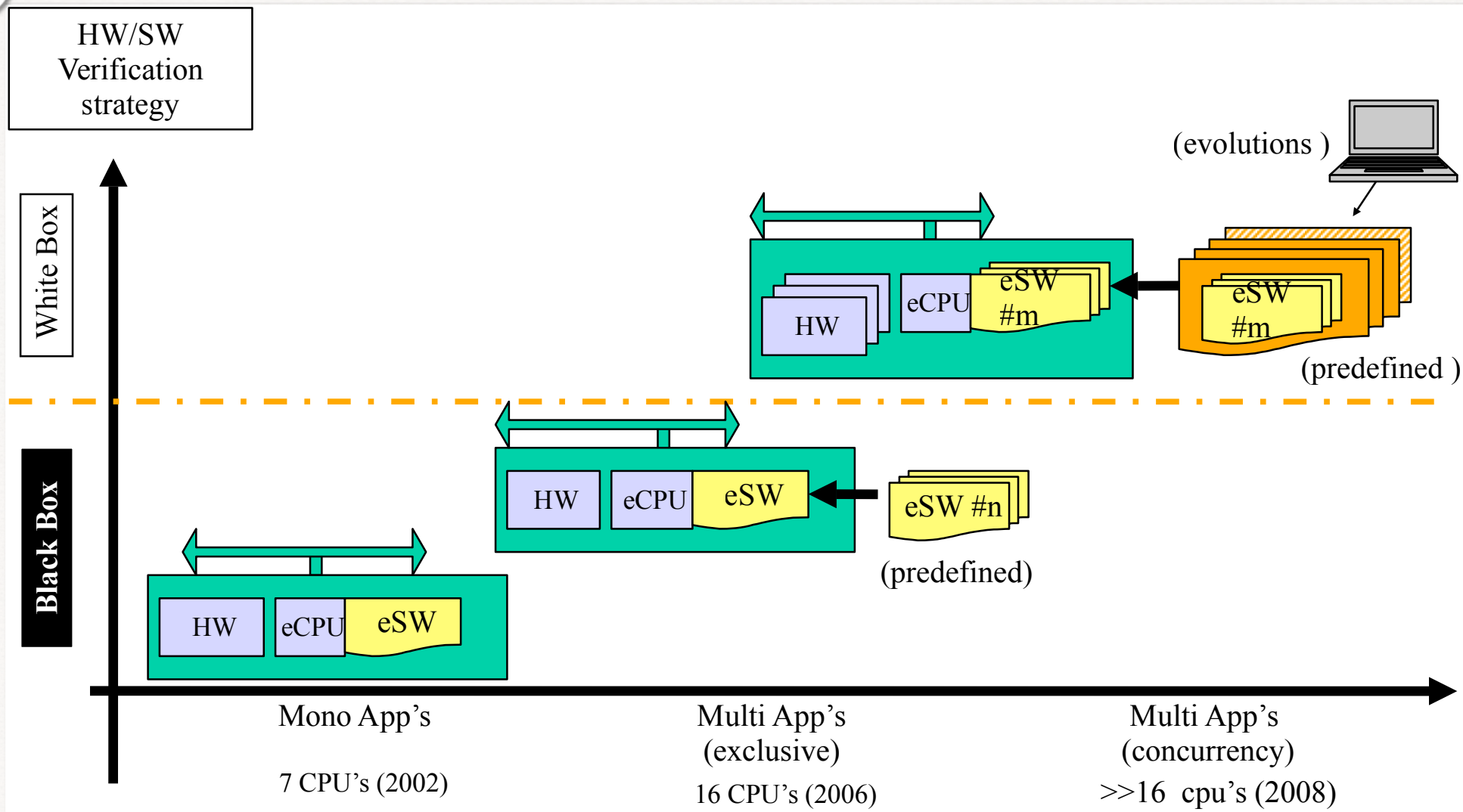


Exemple, le décodeur "H-264" de STMicro

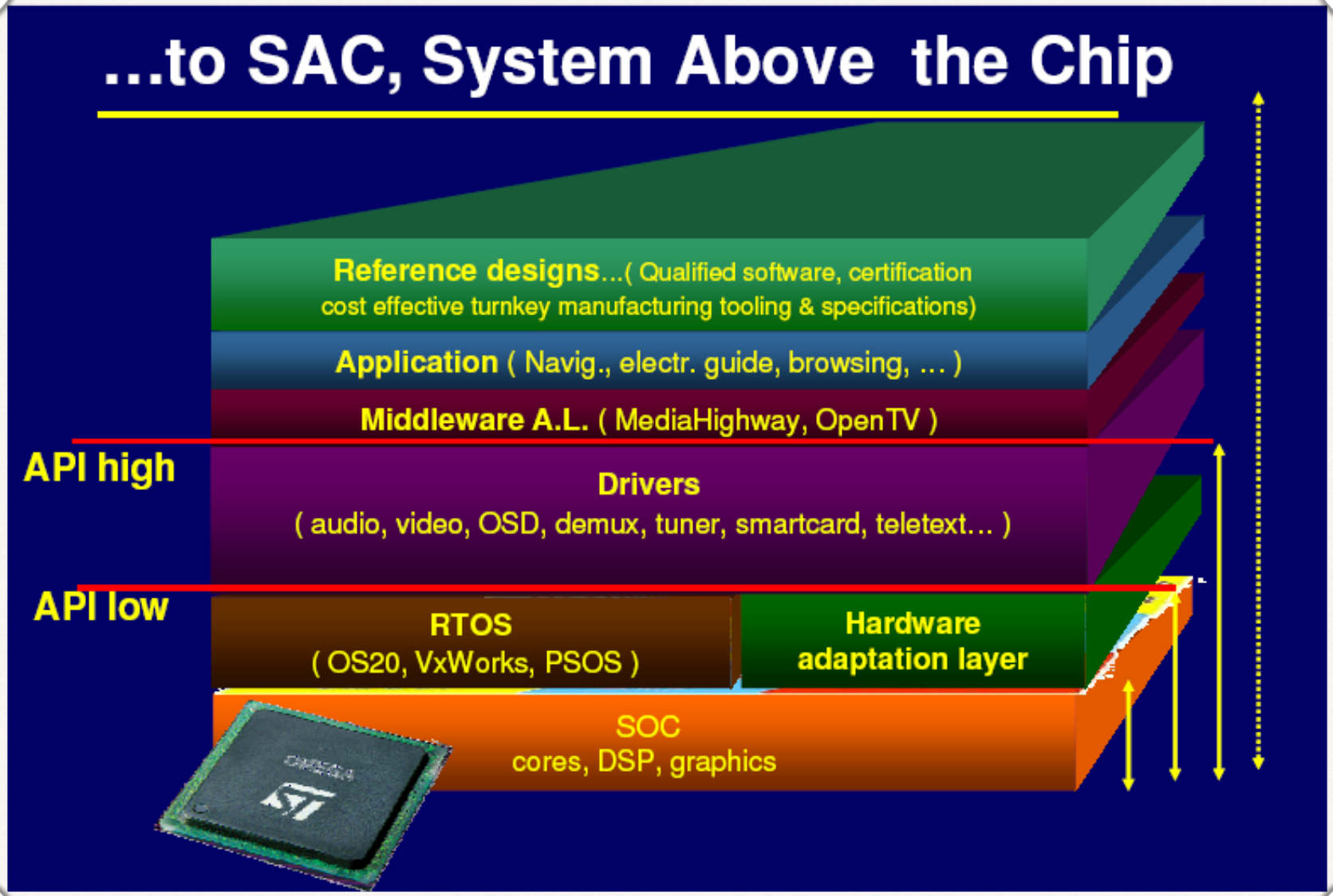


Evolution de la complexité des étapes de vérification

Evolution de la complexité des système a vérifier

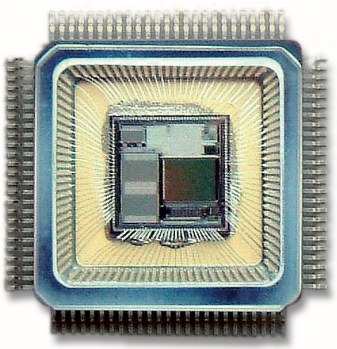


Il existe différents niveaux de couches à vérifier

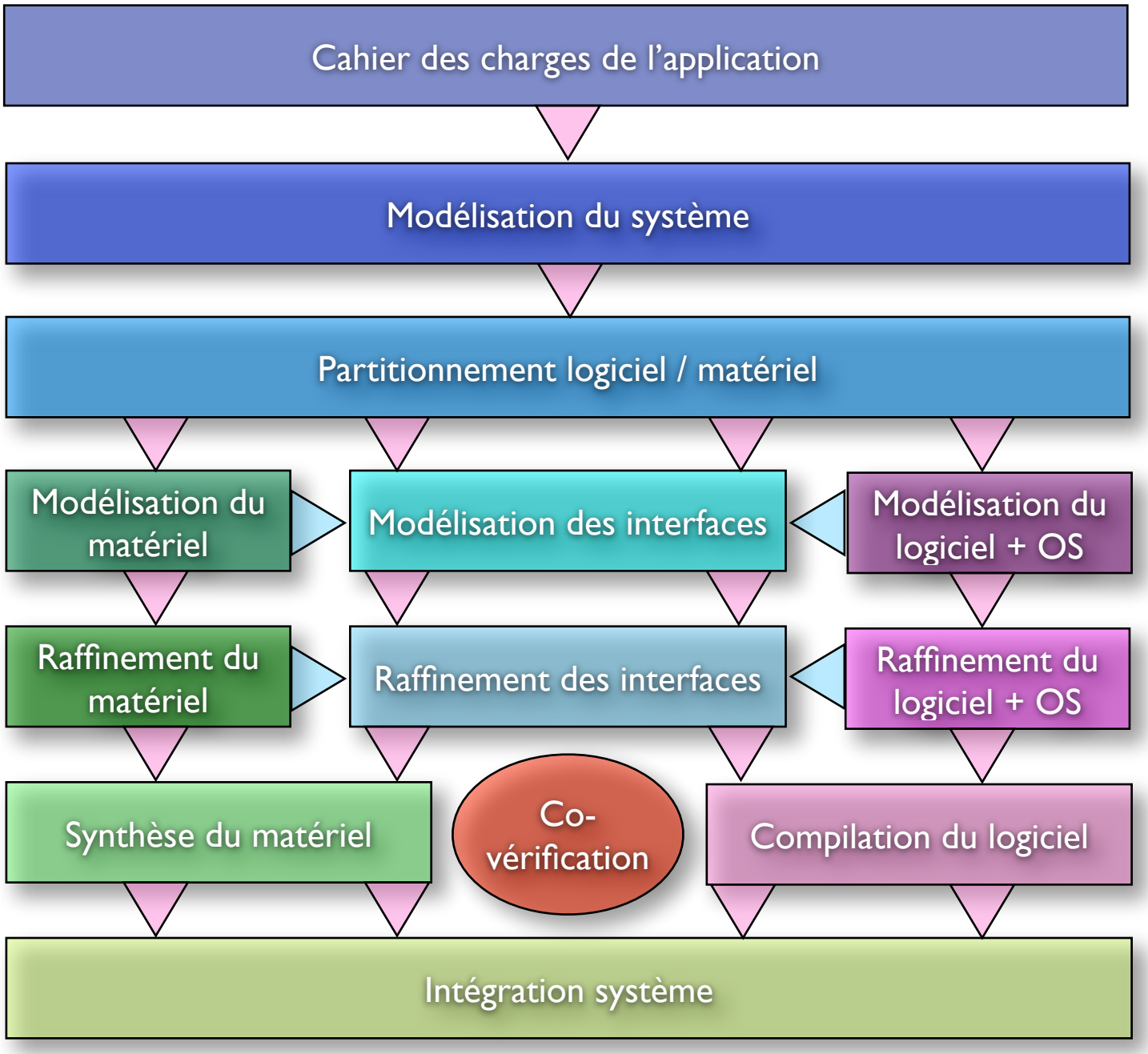


Quel(s) niveau(x) du flot de conception doit on vérifier ?

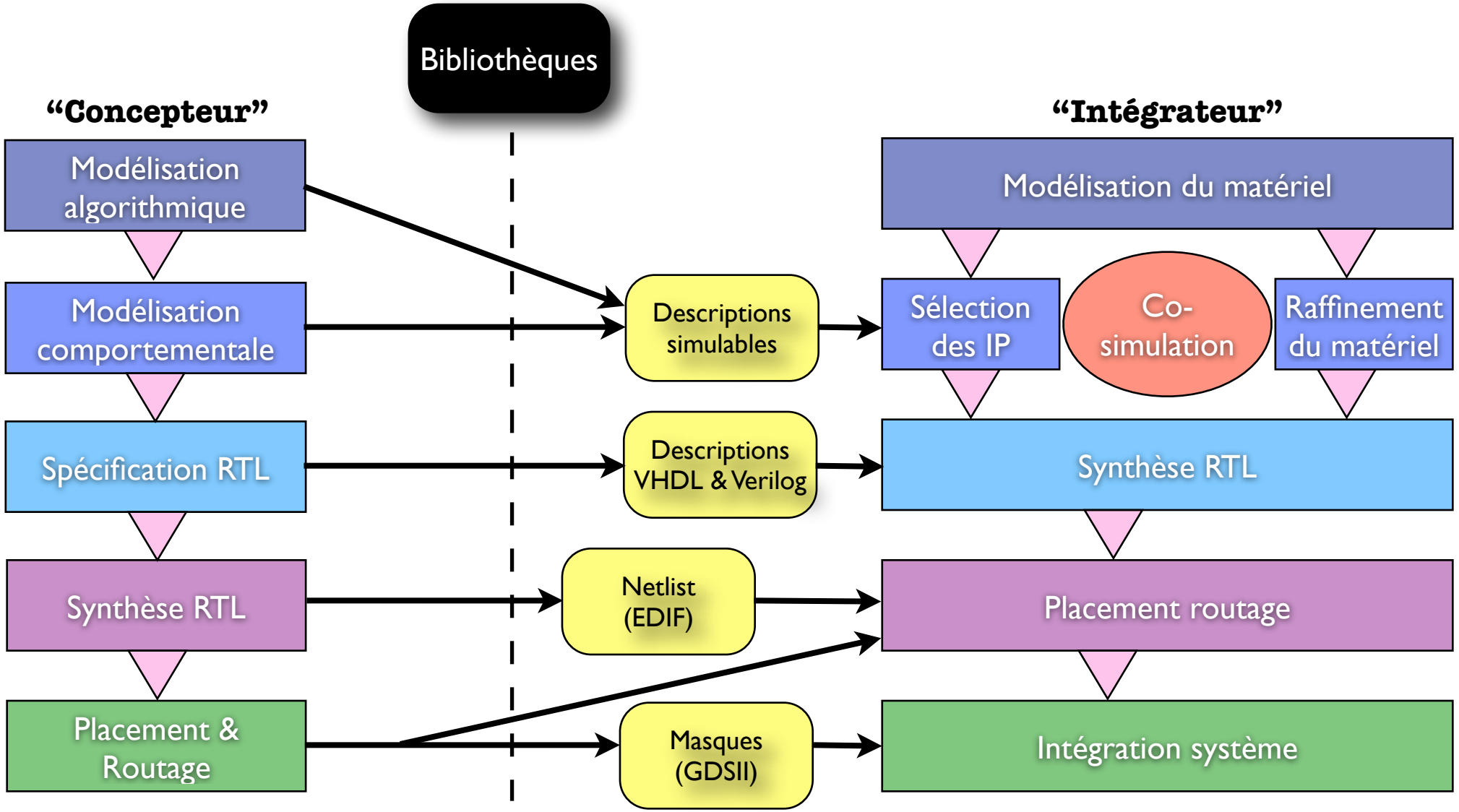
Cahier des charges



Circuit hétérogène



Flot de raffinement matériel



Exemple d'une multiplication flottante (1)

- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- Niveau Untimed (SystemC, VHDL)
- Niveau Timed Functionnal (SystemC, VHDL)
- Niveau Cycle Accurate (SystemC, VHDL)
- Niveau portes (VHDL-RTL)
- Niveau transistors

```
function c = mult(a, b)
c = a * b;
```

Dans le cadre des descriptions comportementales on ne s'occupe que du "comportement" sans se préoccuper du comment (produit de matrices par ex.).

Pas de notion de temps, de dynamique des données, de synchronisation...

Exemple d'une multiplication flottante (2)

- Niveau Algorithmique (MatLab)
- **Niveau Algorithmique (C)**
- Niveau Untimed (SystemC, VHDL)
- Niveau Timed Functionnal (SystemC, VHDL)
- Niveau Cycle Accurate (SystemC, VHDL)
- Niveau portes (VHDL-RTL)
- Niveau transistors

```c
float mult(float a, float b)
{
    int c = a * b;
    return c;
}
```

Dans le cadre des descriptions comportementales en langage C on précise le types des données manipulées ainsi que la manière de réaliser les calculs.

Exemple d'une multiplication flottante (3)

- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- **Niveau Untimed (SystemC, VHDL)**
- Niveau Timed Functionnal (SystemC, VHDL)
- Niveau Cycle Accurate (SystemC, VHDL)
- Niveau portes (VHDL-RTL)
- Niveau transistors

```
sc_lv<32> mult(
        sc_lv<32> a,
        sc_lv<32> b
        ){
    sc_lv<32> c;
    c.set(a.get(31) ^ b.get(31), 31);
    // Suite de la description des
    // calculs a realiser...
    return c;
}
```

Lorsque l'on réalise une description non timé, on introduit des informations sur la future implantation du système : dynamique des données, ordre d'arrivée / production des données, etc.

Exemple d'une multiplication flottante (4)

- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- Niveau Untimed (SystemC, VHDL)
- **Niveau Timed Functionnal (SystemC, VHDL)**
- Niveau Cycle Accurate (SystemC, VHDL)
- Niveau portes (VHDL-RTL)
- Niveau transistors

```
sc_lv<32> mult(
        sc_lv<32> a,
        sc_lv<32> b
        ){
    sc_lv<32> c;
    c.set(a.get(31) ^ b.get(31), 31);
    // ... ... ... ... ... ...
    wait( 100, SC_NS );
    return c;
}
```

Lorsque l'on réalise une description timé, on introduit des informations sur le comportement temporel du système en ajoutant des "temps de calcul" afin de vérifier le respect des contraintes.

Exemple d'une multiplication flottante (5)

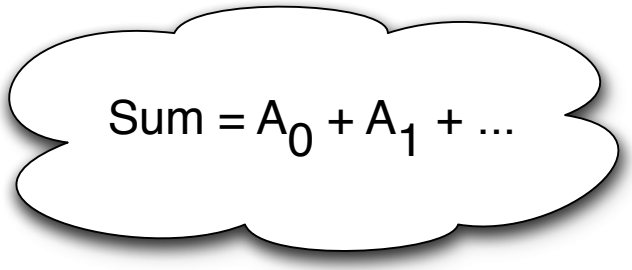
- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- Niveau Untimed (SystemC, VHDL)
- Niveau Timed Functionnal (SystemC, VHDL)
- **Niveau Cycle Accurate (SystemC, VHDL)**
- Niveau portes (VHDL-RTL)
- Niveau transistors

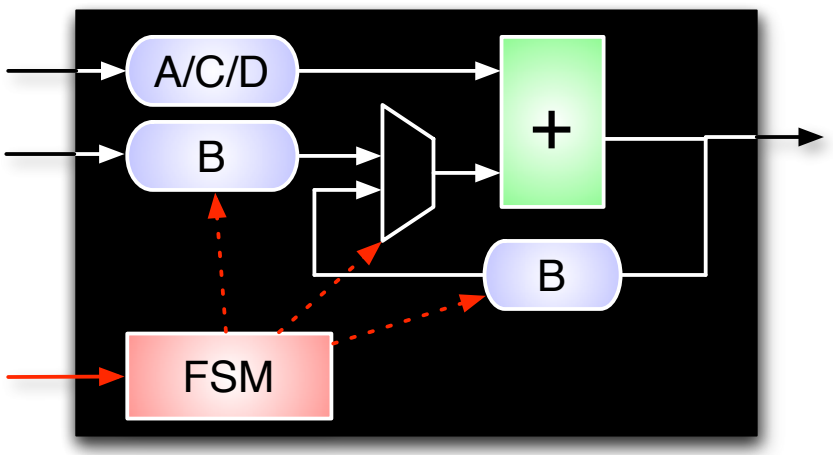
```
sc_lv<32> mult(
        sc_lv<32> a,
        sc_lv<32> b
        ){
   sc_lv<32> c;
   c.set(a.get(31) ^ b.get(31), 31);
   wait( 10, SC_NS );
   // ... ... ... ... ... ...
   // ... ... ... ... ... ...
   wait( 10, SC_NS );
   // ... ... ... ... ... ...
   // ... ... ... ... ... ...
   wait( 10, SC_NS );
   return c;
}
```

Description temporelle des calculs et des transferts de données dans les cycles d'exécution.

Exemple d'une multiplication flottante (6)

- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- Niveau Untimed (SystemC, VHDL)
- Niveau Timed Functionnal (SystemC, VHDL)
- Niveau Cycle Accurate (SystemC, VHDL)
- **Niveau portes (VHDL-RTL)**
- Niveau transistors

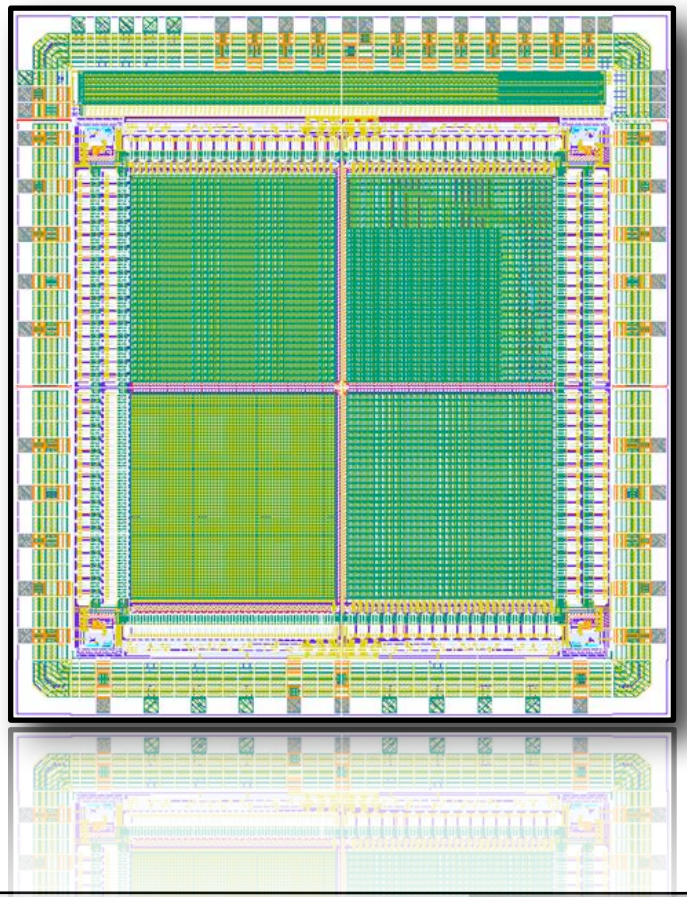




Exemple d'une multiplication flottante (7)

- Niveau Algorithmique (MatLab)
- Niveau Algorithmique (C)
- Niveau Untimed (SystemC, VHDL)
- Niveau Timed Functionnal (SystemC, VHDL)
- Niveau Cycle Accurate (SystemC, VHDL)
- Niveau portes (VHDL-RTL)
- Niveau transistors

Au niveau masque les ressources sont placées et routées, les temps. L'ensemble des caractéristiques est connu avec précision.



Conclusion sur le flot de conception global

- La conception d'un composant matériel ou logiciel sous contraintes est une étape longue et fastidieuse,
  - ➡ On intègre les contraintes et les choix de conception au fur et à mesure,
  - ➡ On doit valider la fonctionnalité du système durant toutes ces étapes,
  - ➡ Valider la cohérence des modèles lors des changements de langage,
  - ➡ Respect des contraintes imposées : cadence, latence, consommation d'énergie, etc.
- Obtenir un circuit ou programme fonctionnant correctement du premier coup tient de l'utopie,
  - ➡ Alors intégrer des dizaines de composants au sein d'un système ne produira assurément pas un système fonctionnel du premier coup...
- Il est nécessaire de vérifier les différents raffinements réalisés lors des phases de conception !

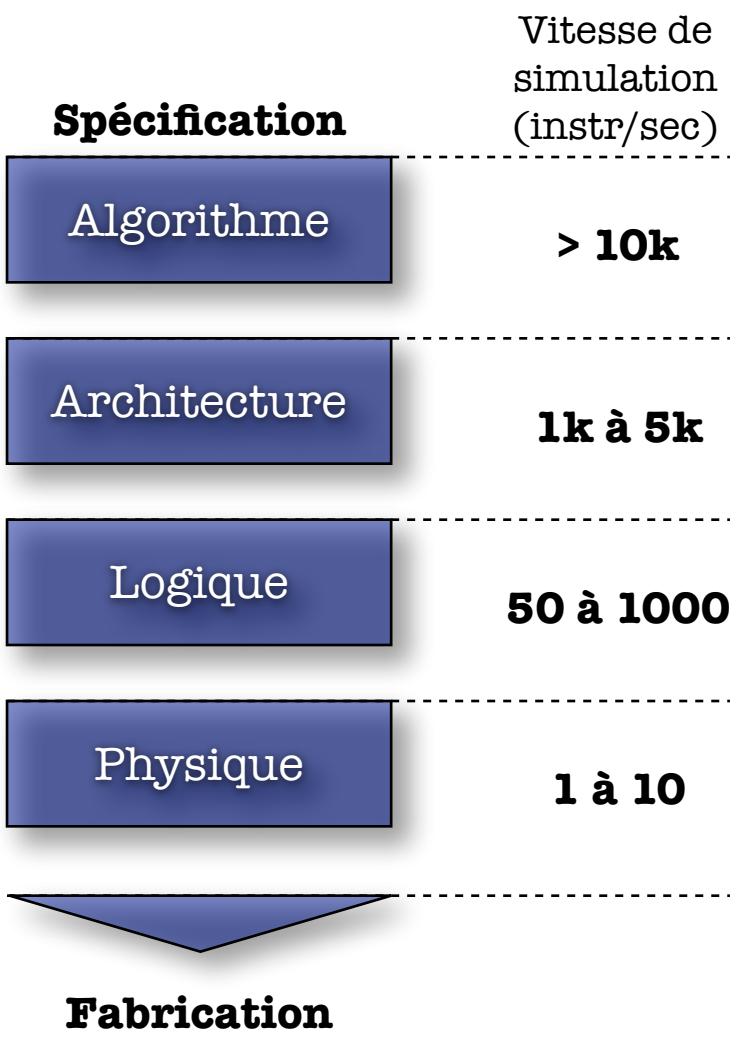
Les raisons de la vérification

Le lien entre les couts et la détection des problèmes

SoC…

Si erreur au niveau	Argent perdu	Time to market perdu
système	5.000 $	3 minutes
RTL	5 M $	3 jours
circuit	100 M $	3 mois

L'évolution de la vitesse de simulation des modèles



En fonction des niveaux de description utilisés lors de la conception, le temps de simulation varie :

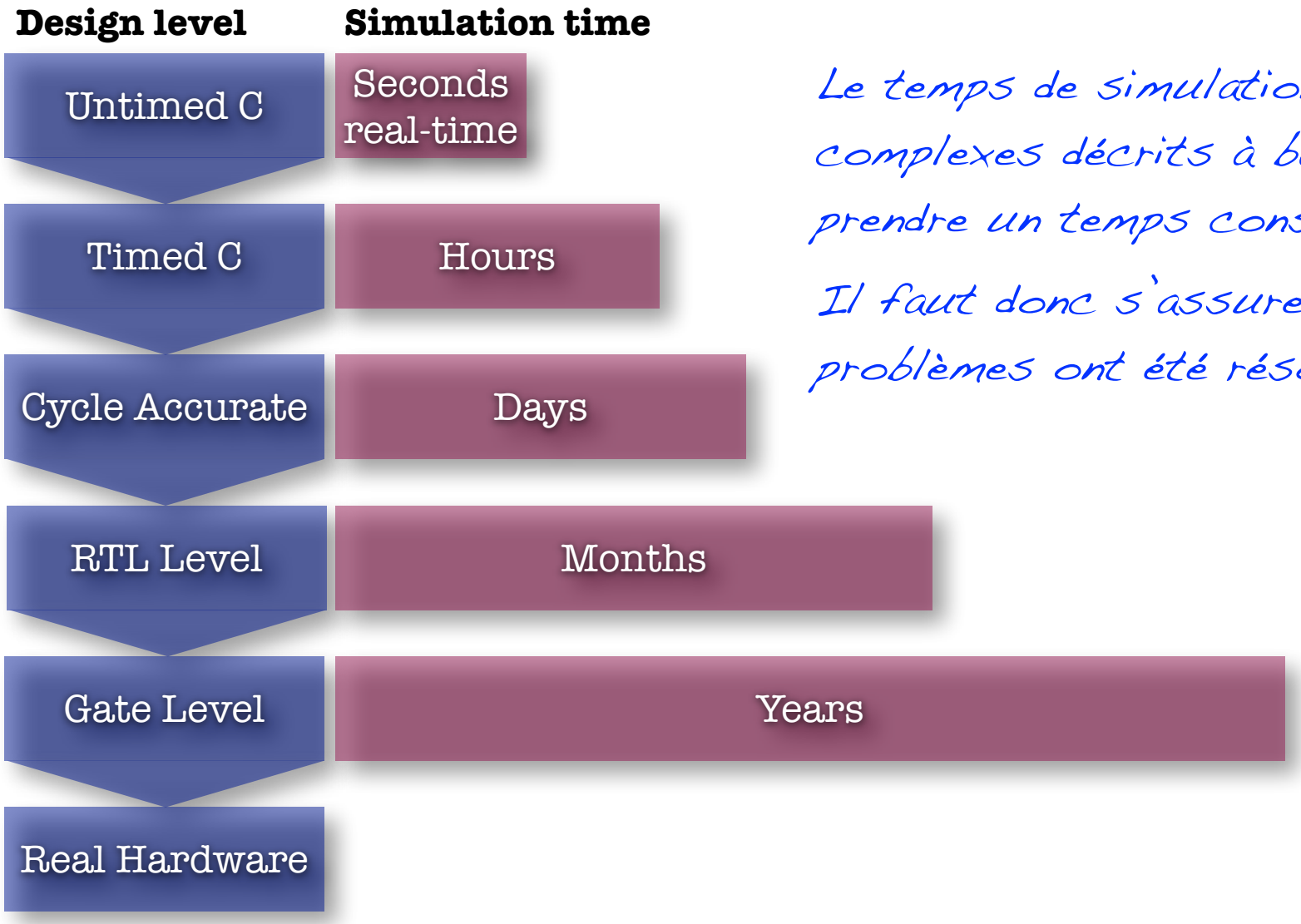
=> Plus de paramètres à considérer,

=> La taille des descriptions à simuler varie

=> etc...

Il suffit déjà de comparer le temps de simulation d'un programme écrit en langage "C" par rapport à un programme écrit en assembleur.

Temps nécessaire à la simulation de systèmes complexes



Le temps de simulation de systèmes complexes décrits à bas niveau peut prendre un temps considérable...

Il faut donc s'assurer que les problèmes ont été résolus auparavant !

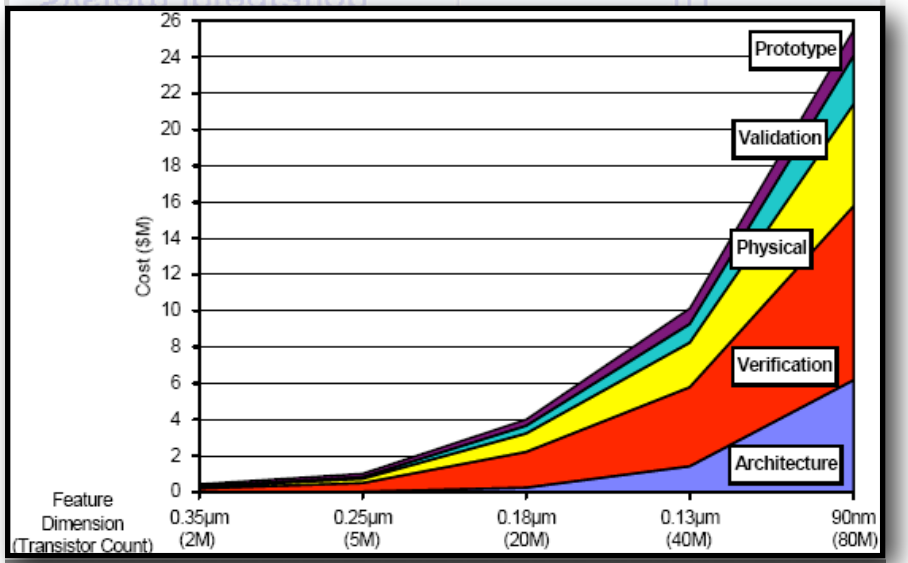
Et la vérification dans l'apprentissage des langages ?

- Dans la majorité des livres sur les langages HDL, on ne s'intéresse qu'à la syntaxe et à la sémantique des langages.
- L'objectif est d'écrire :
  - ➡ Compilable (ModelSim),
  - ➡ Simulable (ModelSim),
  - ➡ Synthétisable (ISE, Quartus),
- La partie adressant la vérification est remise à plus tard…
  - ➡ Cela ne doit donc pas être très important !

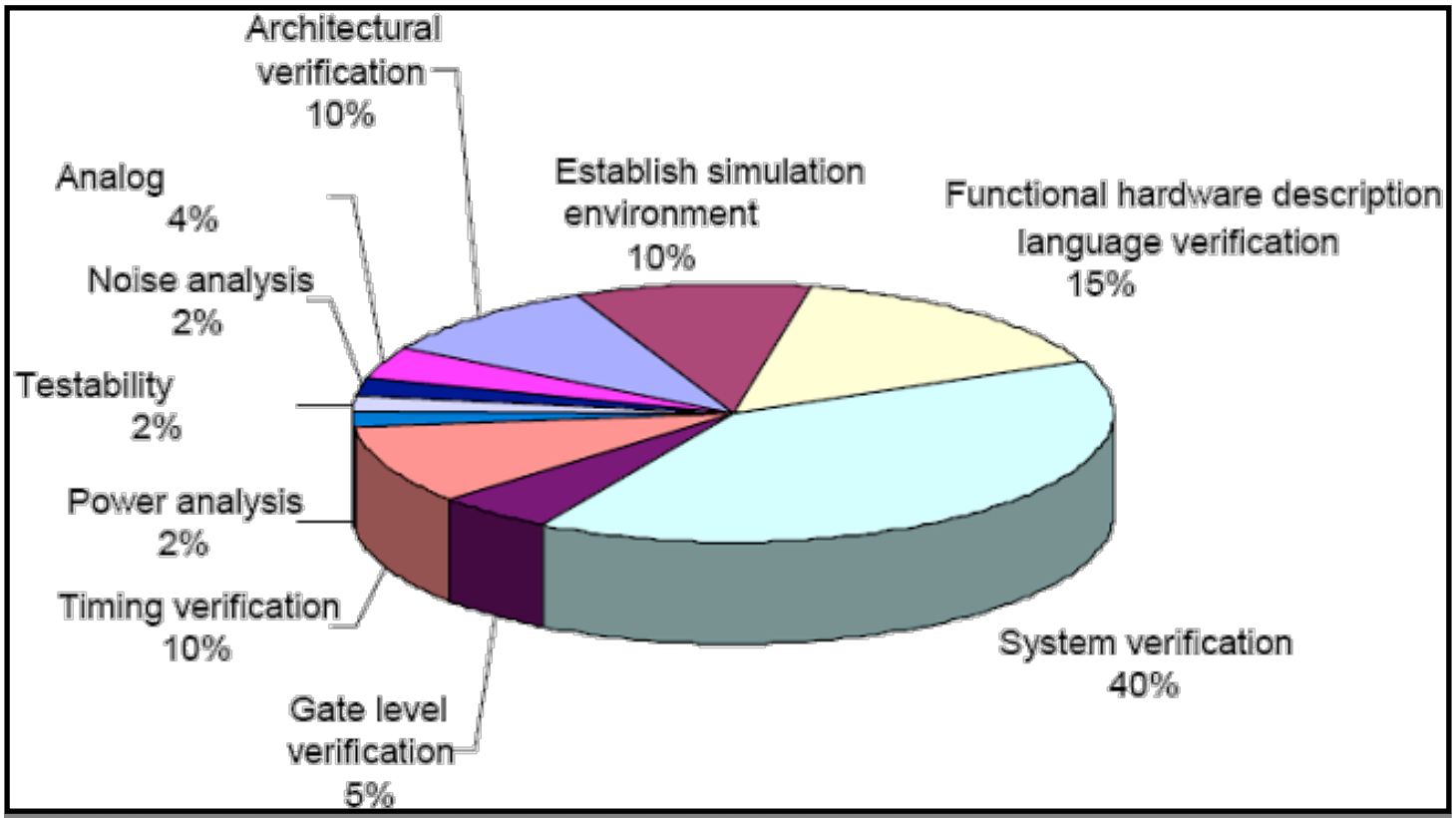
Qu'en est il dans la réalité des équipes de conception ?

- La réalité est un peu différente
  - Le temps dédié à la vérification lors de la conception d'un système complexe (SoC) peut atteindre 70% du temps total de conception,
  - Les équipes dédiées à la vérification sont plus importantes (en nombre de personnes) que celles dédiées à la conception "pure",
- « *Everyone knows debugging is twice as hard as writting a program in the first place* », Brian Kernighan, Elements of programming style, 1974

Tasks	Time Ratios (%)
Verification	40
RTL Synthesis	20
IC Layout	10
Test	10
System Integration	10
Other	10



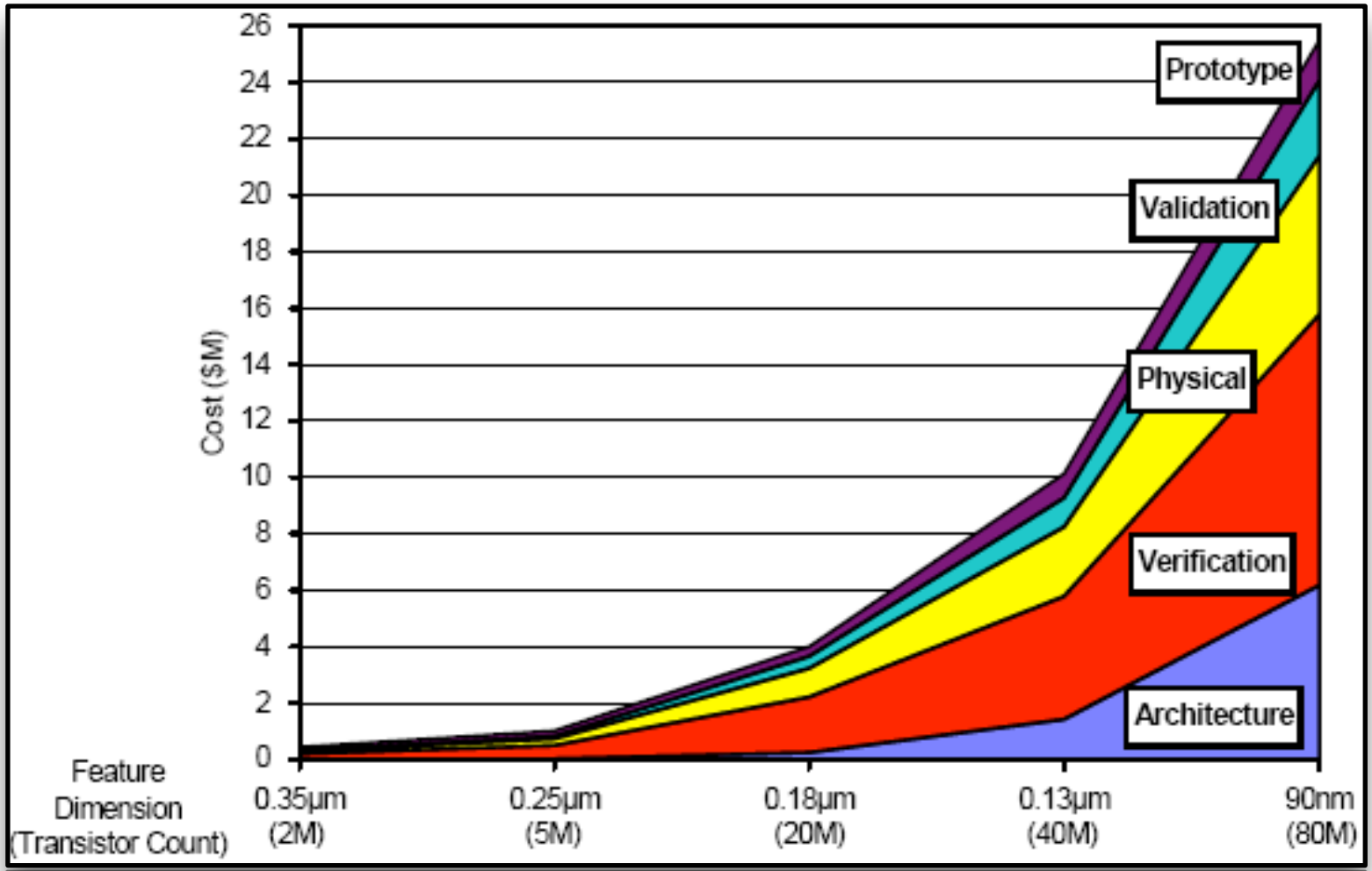
Quels sont les points clefs que l'on vérifie ?



Répartition temporelle des différentes tâches dédiées à la vérification

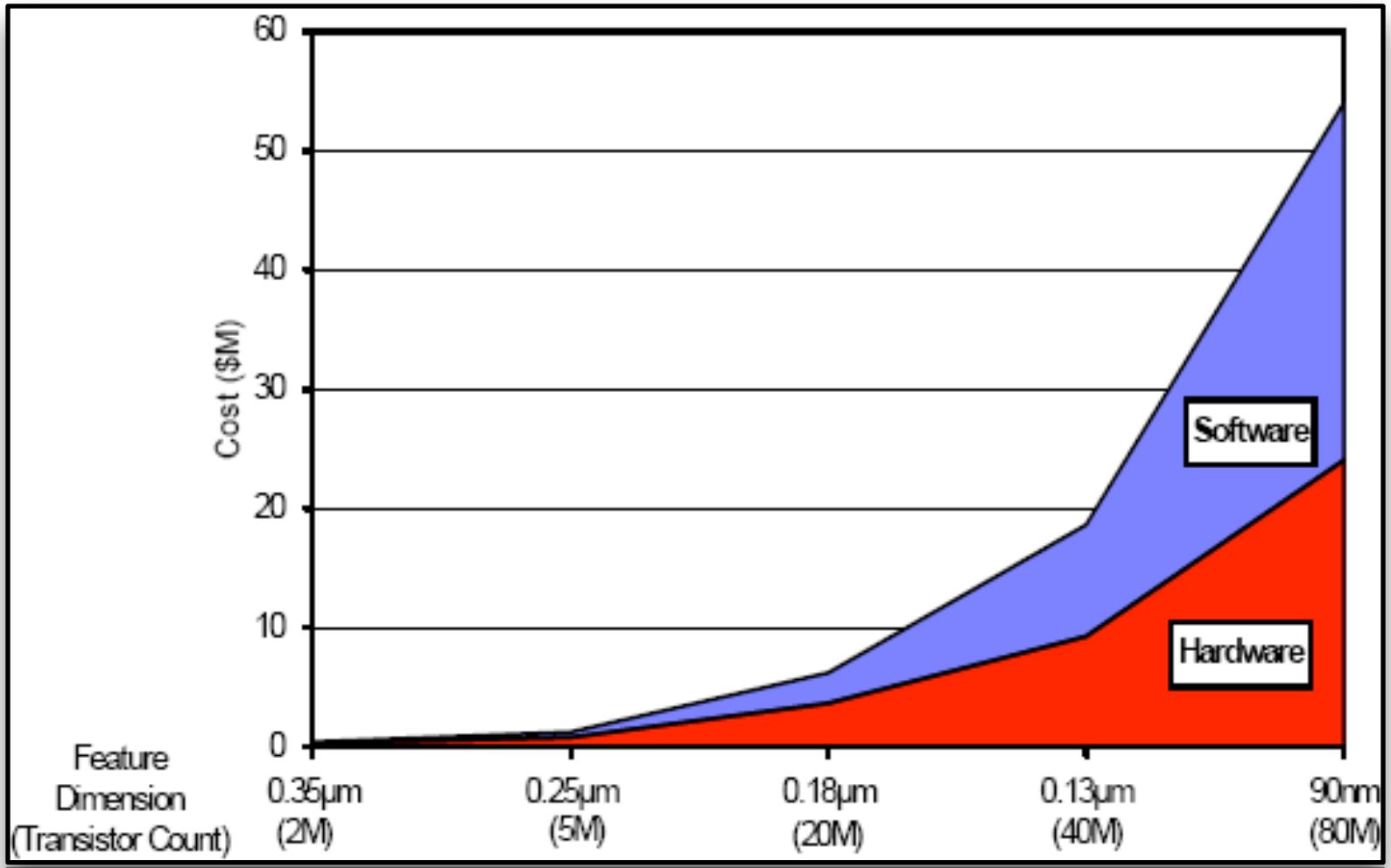
[Source : ITRS 2003]

Répartition des couts d'étude et de conception



ST Microelectronics (L. DUCOUSSO) - GDR SOC-SIP – 14 juin 2007

Evolution du cout de la vérification



ST Microelectronics (L. DUCOUSSO) - GDR SOC-SIP – 14 juin 2007

Les contradictions liées aux processus de vérification

- La vérification est un mal nécessaire qui :
  - Coûte beaucoup trop cher ($),
  - Dure trop longtemps (Time to Market),
  - N'est pas à proprement parlé générateur d'argent,
  - Est ingrat pour les concepteurs (jamais valorisant de trouver ses erreurs).
- La vérification est un processus jamais vraiment fini :
  - On peut prouver la présence d'erreurs, mais pas leur absence !
  - Plus le temps passe, moins il reste d'erreurs et plus le temps passé pour les détecter croit…
- L'erreur que l'on va peut être trouver est elle assez sévère pour investir du temps (et de l'argent) pour la corriger ?

Introduction à la notion de plan de vérification

Comment organiser le processus de vérification ?

- Question fondamentale
  - Comment organiser la vérification (agenda, ressources humaines, coût) ?
- Avant de commencer le développement d'un produit, il est nécessaire de :
  - Savoir ce que l'on va devoir vérifier pour garantir la fonctionnalité souhaitée,
  - Savoir quand on va pouvoir (devoir) vérifier la fonctionnalité visée,
  - Que va-t-on considérer comme étant la première version correcte du circuit ?
- Ces problématiques doivent être traitées à partir des spécifications de référence. Cela abouti au plan de vérification.
- Remarques
  - L'ordre "normal" des choses => (1) spécification, puis (2) implémentation,
  - La spécification ne doit pas dépendre de l'implémentation (comment vérifier sinon puisque la spécification changera à chaque fois que l'implémentation changera ?)

Des spécifications aux fonctionnalités à vérifier

- Point de départ : les spécifications
  - Il faut identifier les propriétés à vérifier : d'abord les interfaces, puis les fonctionnalités à valider.
- Propriétés de chaque interface ? Exemple de questions à se poser pour y répondre :
  - Quelles séquences envoyer ?
  - Quel est l'ensemble des valeurs ?
  - Quelles sont les séquences de transactions possibles ?
  - Quelles violations du protocole doit on détecter ?
  - Besoin de synchroniser le protocole de cette interface avec celui d'une autre interface ?
  - …

Des spécifications aux fonctionnalités à vérifier

- Ordonner par priorité les propriétés à vérifier : d'abord les + importantes (…jusqu'aux moins prioritaires, qui recevront moins d'attention, voir qui ne seront pas traitées si pas le temps (propriétés à vérifier si possible ≅ optionnelles))
  - ➡ Vérifier seulement ce qui est nécessaire…
- Identifier les propriétés difficiles à vérifier.
- Modifier le design pour faciliter leur vérification .
- Certaines fonctionnalités peuvent impacter sur d'autres...
  - ➡ Bien ordonner les priorités de vérification permet de gagner en efficacité et donc en temps !

Des spécifications aux fonctionnalités à vérifier

- Vérifier les bancs de tests, comment s'assurer de la validité des testbenchs écrits ?
  - ➡ Faire vérifier ces derniers par des personnes neutres,
  - ➡ En utilisant une couverture fonctionnelle (Functionnal Coverage),
- L'intérêt des cas d'utilisation est de fournir une mesure de l'avancement du projet,
- En fonction du nombre de cas d'utilisation déjà validés, il est possible d'estimer l'avancement du projet.
- Les circuits contiennent des millions de portes et des dizaines de fonctionnalités il est utopique de penser qu'ils sont bug-free !

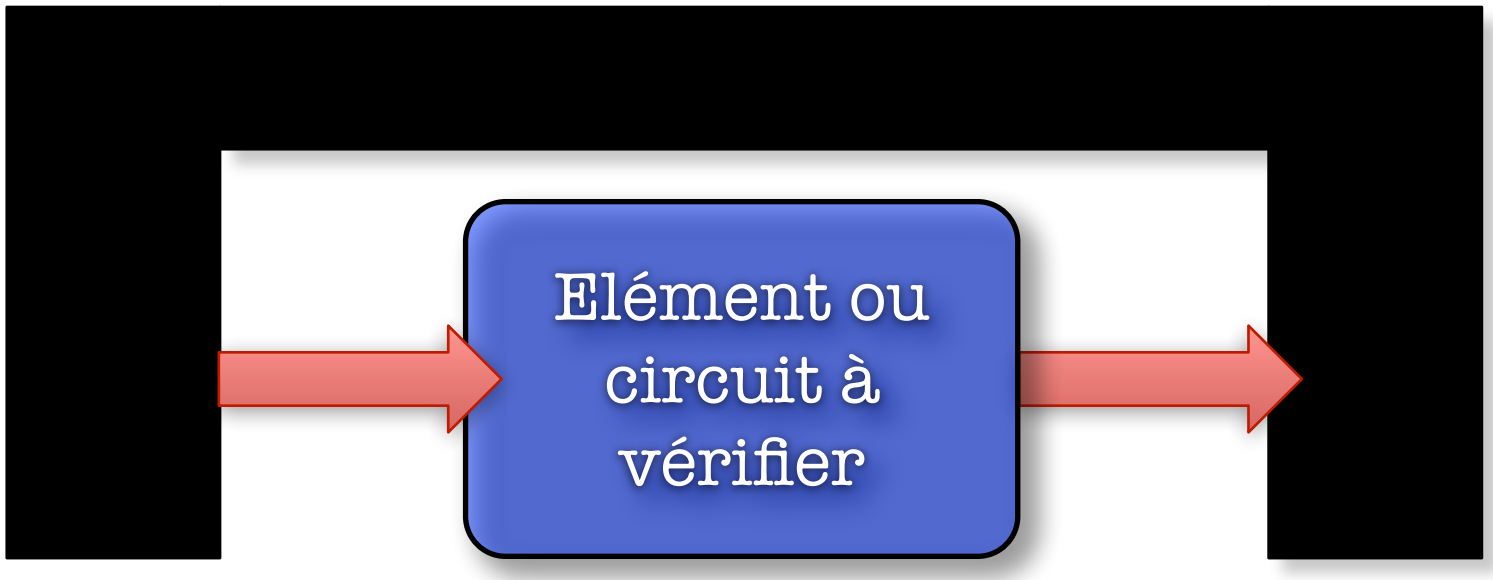
La contradiction en l'optimisation <=> la vérification

- Priorité lors de la conception
  - La maintenabilité est un point clef lorsque l'on écrit du code synthétisable (importance de bien commenter son code source),
  - Si le design une fois terminé respecte les contraintes imposées par le cahier des charges, il ne faut pas l'optimiser !
- Optimiser les bonnes choses
  - Réduire le nombre lignes de code n'est pas intéressant économiquement parlant (sauf afin d'augmenter la maintenabilité du code source),
  - Optimiser les performances d'un code implique obligatoirement un coût,
    - Coût financier (temps passé par les concepteurs),
    - Baisse de la maintenabilité (optimisations particulières <=> cas particuliers),
- Conclusion
  - On n'optimise uniquement que lorsque cela est vraiment nécessaire !

Comment vérifie t'on un circuit ?

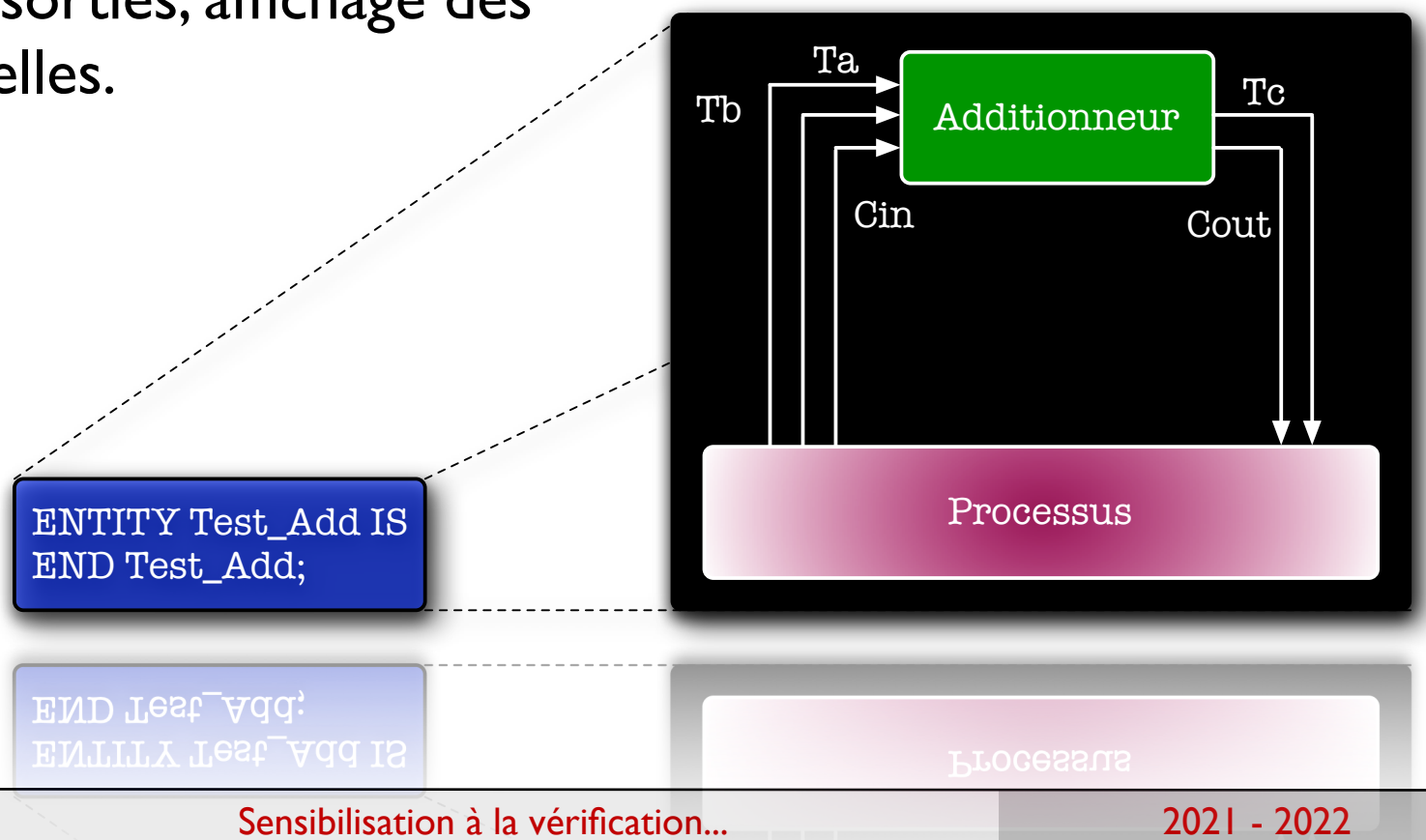
La manière usuelle...

- On entend généralement par « testbench » l'ensemble des bouts de code qui permet d'appliquer un jeu d'entrées prédéterminé à un "composant" afin d'observer sa réponse.



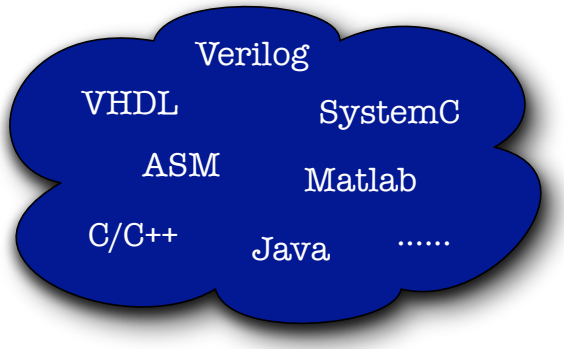
Qu'est ce qu'un banc de test ?

- Instanciation du composant à tester
- Initialisation des signaux d'entrées
- Application d'une séquence de stimuli
- Analyse des résultats, analyse des transitions des sorties, affichage des erreurs éventuelles.

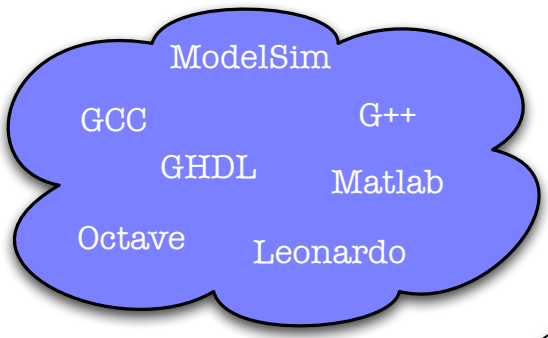


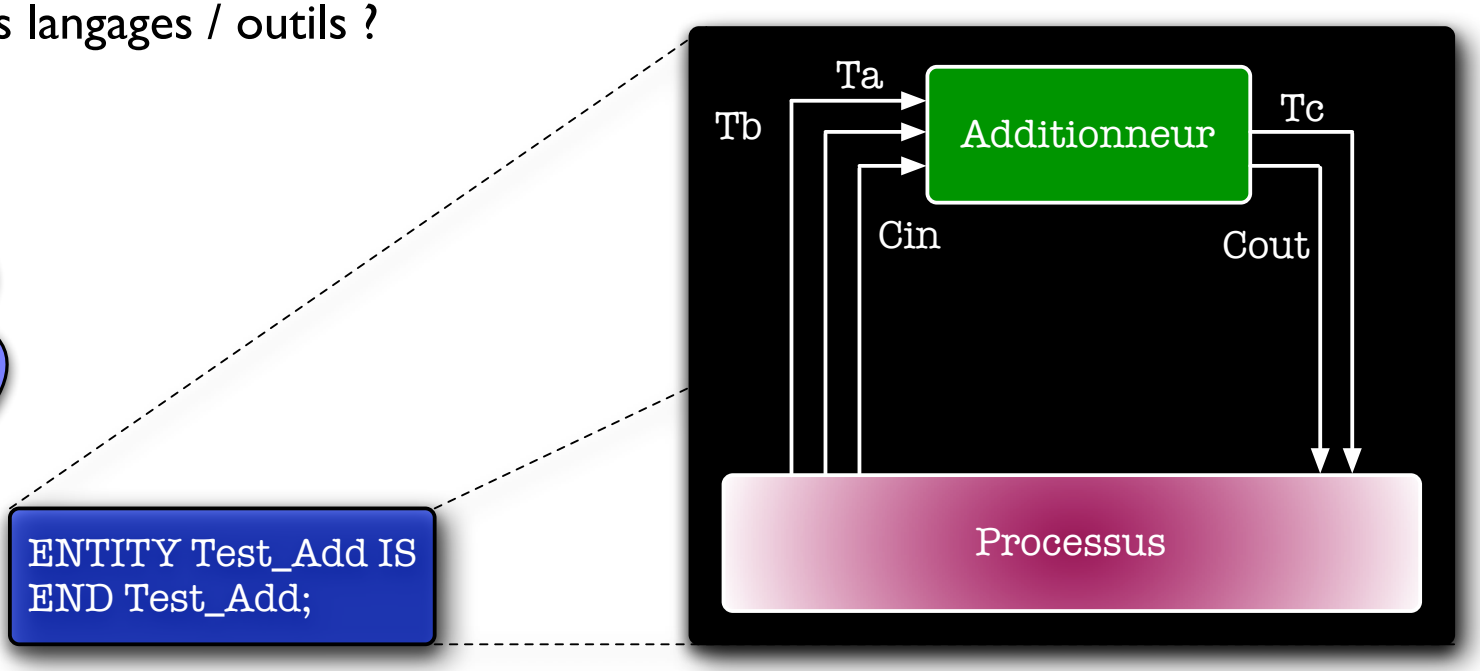
Qu'est ce qu'un banc de test ?

- Les test-benchs doivent être utilisé à chaque étape de raffinement du flot de conception afin de valider la conception.



- Mais pour autant, peut'on :
  - ➡ Utiliser le même test-bench ?
  - ➡ Utiliser les mêmes langages / outils ?

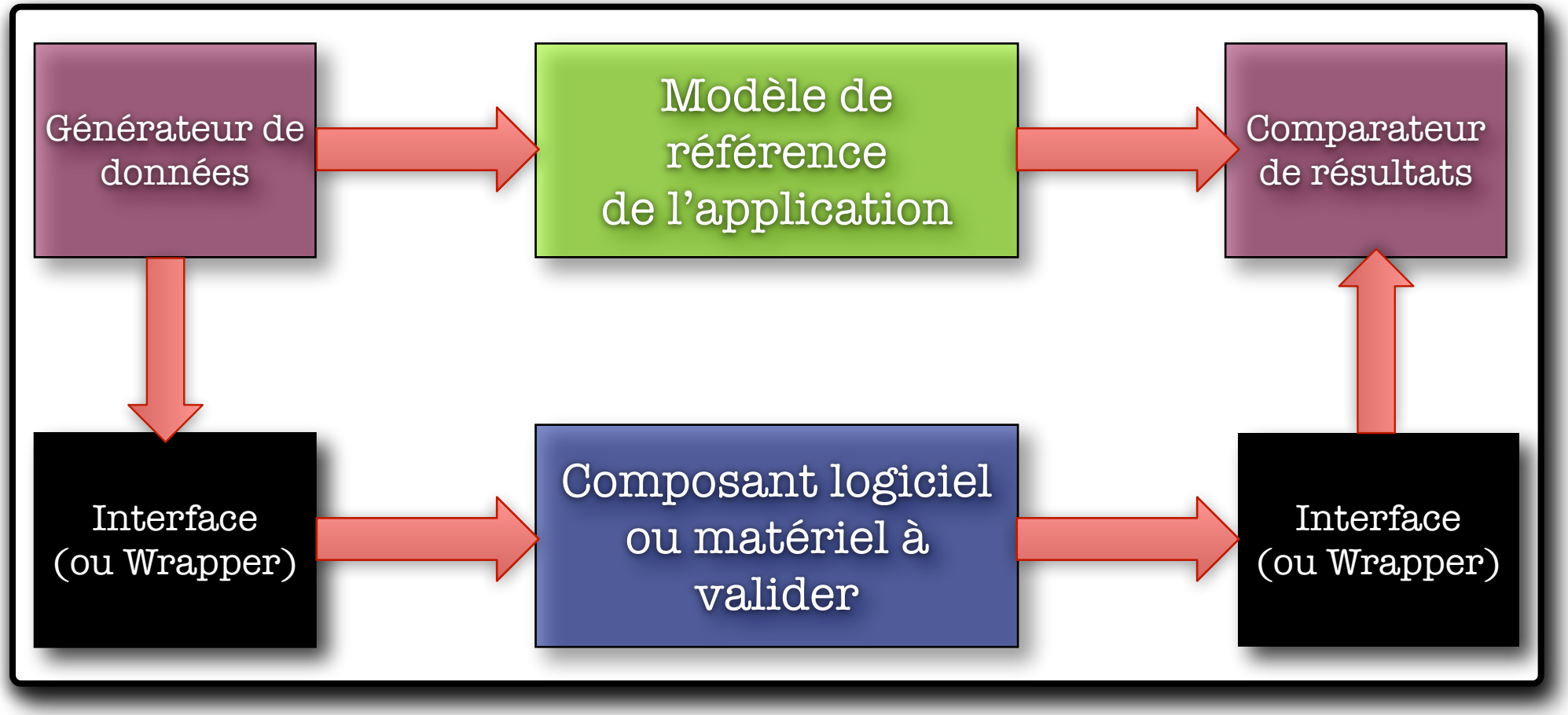




L'architecture typique d'un testbench

- La conversion des données est réalisée dans les interfaces de communication,
  - Evolutivité des bancs de tests au cours de la conception (lors du processus de raffinement, seules les interfaces changent),
  - Diminuer l'intervention humaine autant que possible (gain de temps, limitation du risque d'erreurs),
- Génération des données
  - A la main ? Utilisation d'outils ou de scripts ?
  - Analyse des réponses
- Viser une vérification automatique signalant que les tests ont été passés avec succès ou qu'ils ont échoués,
- Le fait de retirer le harnais de tests ne compromet pas la validation fonctionnelle opérée !

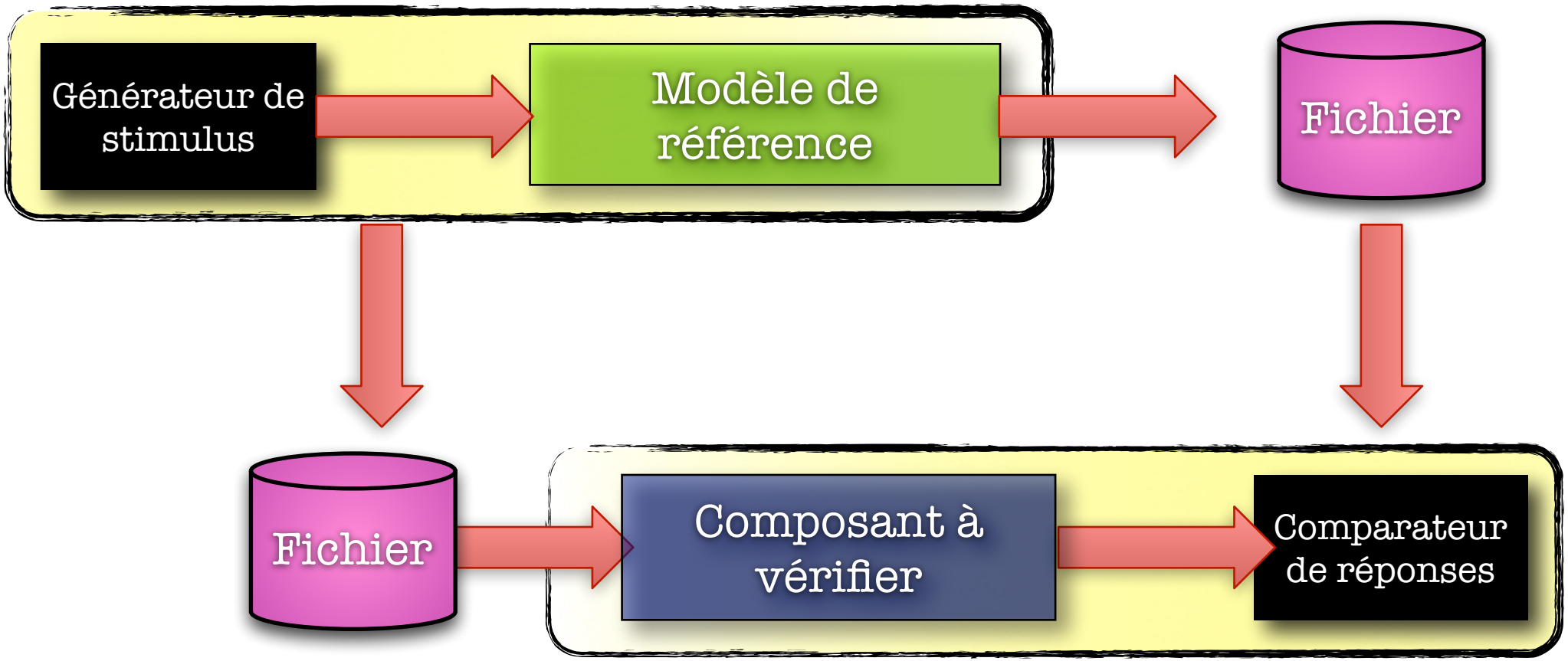
L'idéal est de posséder un modèle de référence ?!



Self-checking testbenches

L'idéal est de posséder un modèle de référence ?!

Approche similaire dans le cas ou plusieurs langages sont utilisés



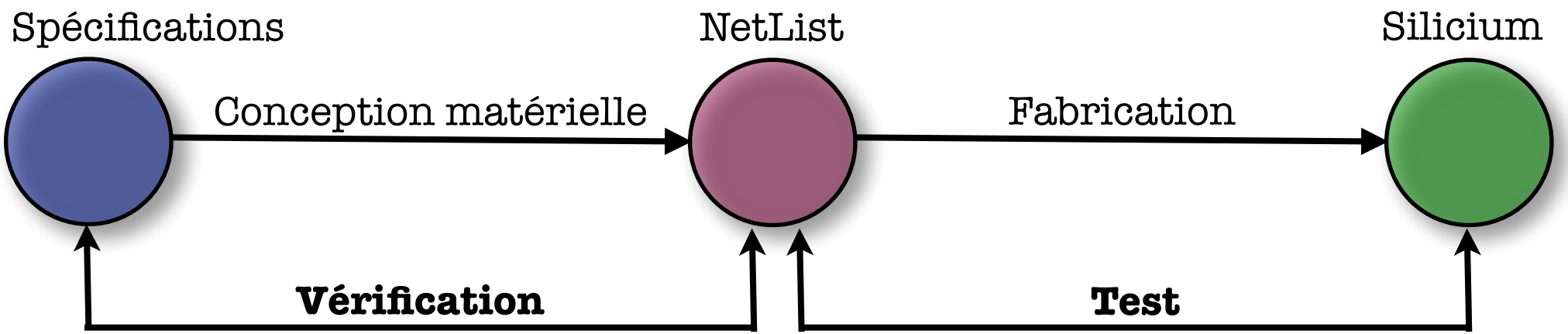
Les bancs de test à validation «automatique»

- La vérification manuelle des résultats obtenus après passage des bancs de test n'est pas une solution acceptable à long terme,
  - ➡ Il est donc nécessaire de déployer des bancs de test qui vérifient les résultats,
  - ➡ Il est important de bien définir ce qui doit être testé,
- La partie de Self-Cheking est une partie complexe à mettre en œuvre :
  - ➡ Stimulus et réponses codées en dur (cela permet de s'assurer des cas d'utilisation qui sont vérifiés),
  - ➡ Génération de données pseudo aléatoire en adéquation avec des parties codées en dur (cas des trames réseau),
  - ➡ Utilisation d'un modèle de référence ou de fonction de transfert (co-simulation avec d'autres modèles, ou spécification de référence),
- L'ensemble de ces points sera étudié plus tard dans la présentation...

Les principes de base de la vérification

Mettons nous d'accord sur la terminologie...

- Commençons par un peu de sémantique...
  - Vérification
    - S'assurer que la conception répond aux objectifs (étape de conception) : on s'assure que les résultats fonctionnels sont corrects.
  - Test
    - S'assurer que la réalisation est conforme à la conception (étape de fabrication). Utilisation de vecteurs de test pour s'assurer que les connections physiques sont correctement réalisées (ATPG : Automatic Test Pattern Generation)

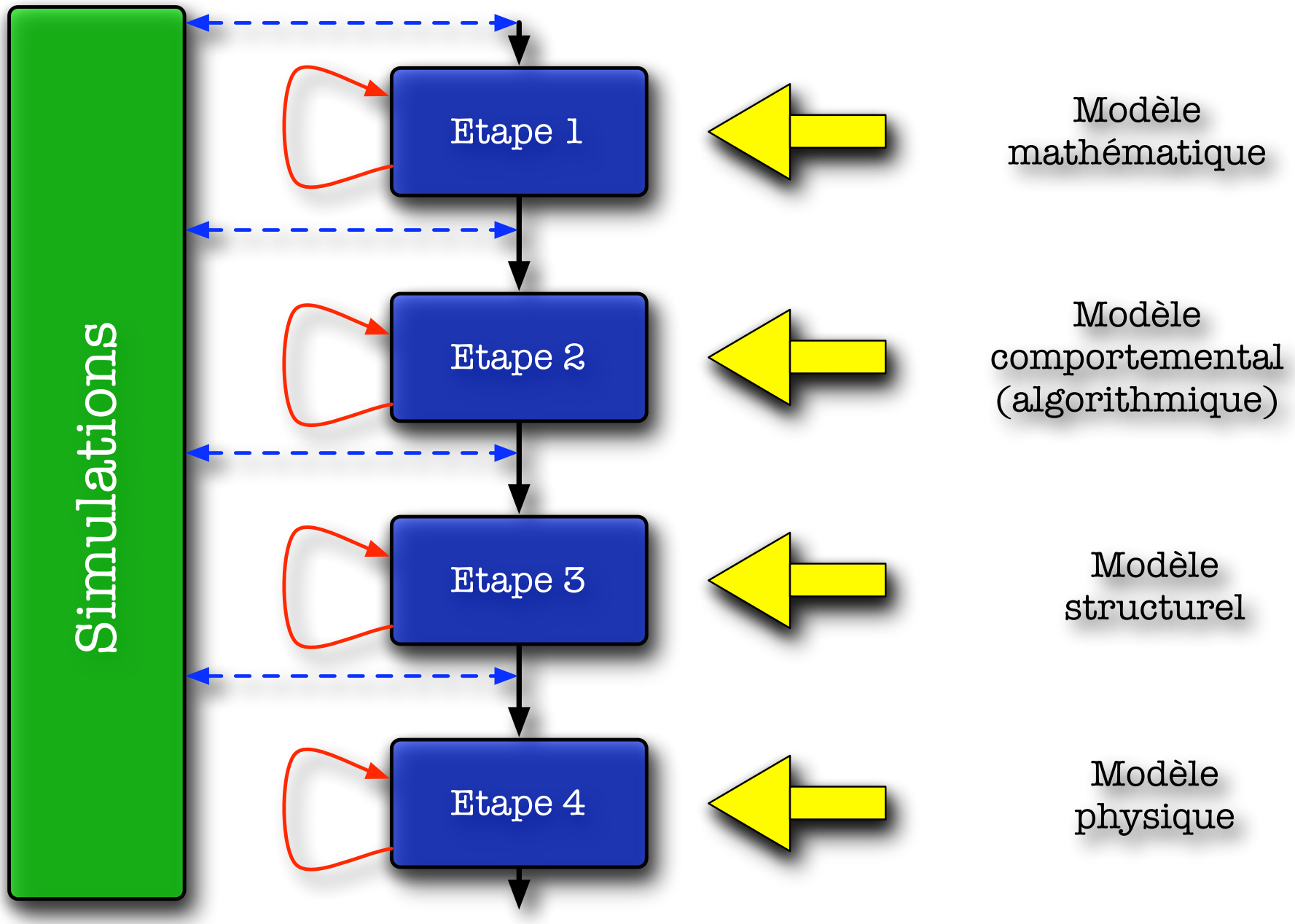


Mais « test, tester, ... » souvent utilisé à la place de « vérification, vérifier, ... » dans le langage courant du concepteur...

Définition du modèle conceptuel de convergence

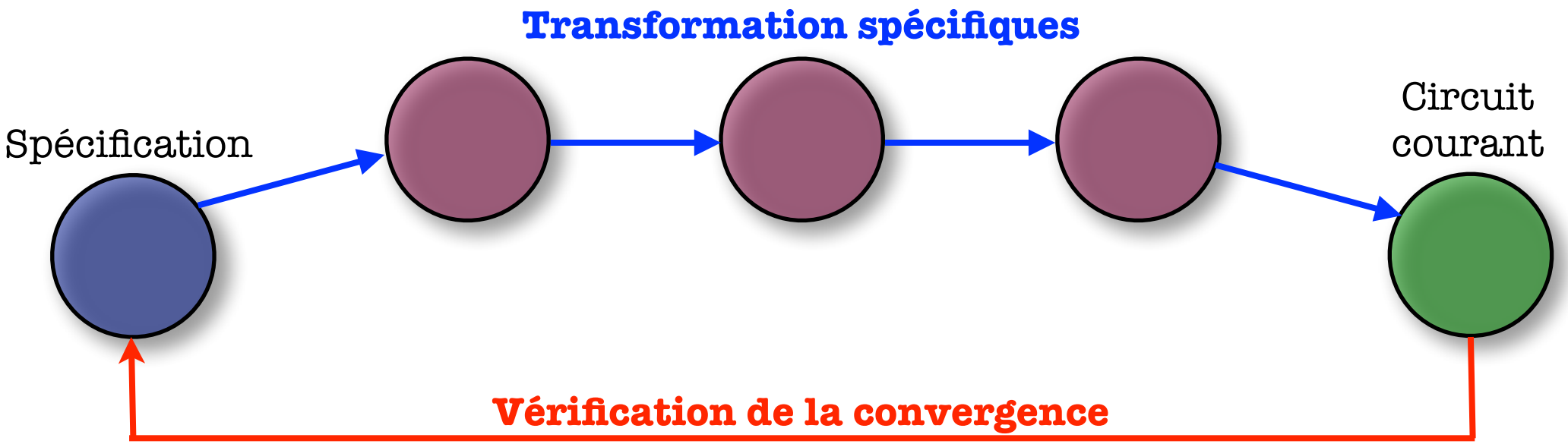
- C'est un modèle conceptuel représentant le processus de vérification utilisé.
- On représente au minimum les points de départ, d'arrivée et la transformation réalisée afin de visualiser ce que l'on vérifie (que vérifie-t-on ?)

Vérification de la convergence dans le flot système



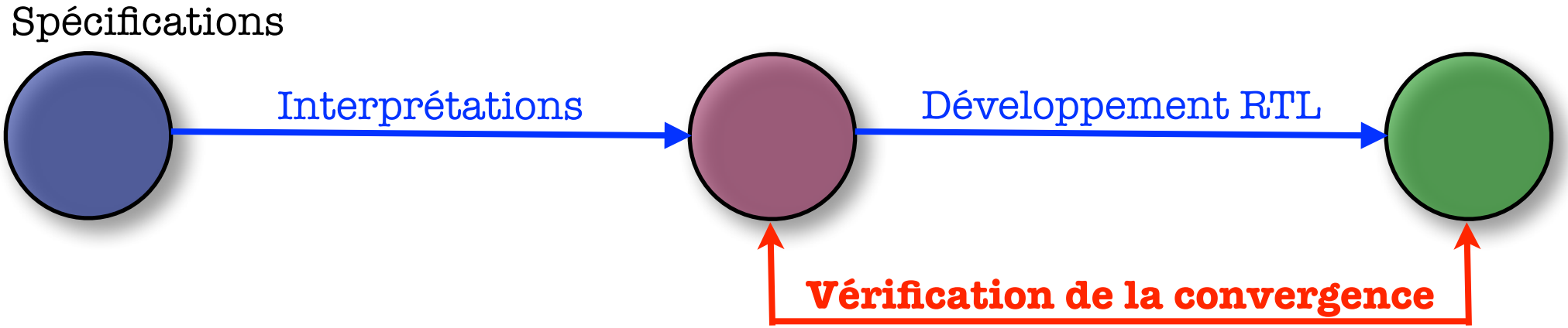
Qu'est ce qui doit être vérifié ?

- Le choix du point d'origine et de reconvergence permet de déterminer ce qui va être vérifié :
  - ➡ Ce sont les différentes transformations et raffinements présents entre ces points qui vont être vérifiés.



Qu'est ce qui doit être vérifié ?

- Si le processus n'est pas entièrement automatisé, cela implique une interprétation des spécifications par une ou plusieurs personnes afin de réaliser certaines transformations.
  - L'erreur est humaine …
- Exemple pour un codage de niveau RTL
  - Une équipe va interpréter une spécification et développer un code RTL en fonction de ce qu'elle perçoit de la spécification
    - Attention aux ambigüités d'interprétation !!!

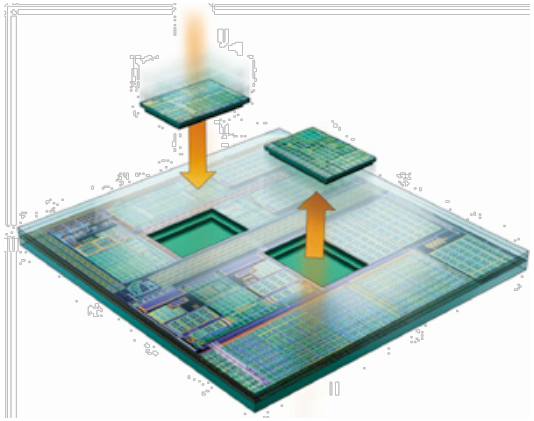


Comment réduire le temps de développement ?

- Pour réduire le temps de développement d'un circuit il est nécessaire de :
  - ➡ Développer et de vérifier le circuit en même temps,
  - ➡ Travailler (développer et vérifier) autant que possible à haut niveau d'abstraction (cela est plus rapide et moins complexe car moins de détails),
  - ➡ Essayer d'automatiser le plus possible la phase de développement du circuit
    - Les phases de développement et de raffinement
    - Phases de vérification des composants
- *Certains outils proposent pour des flots de conception logiciels permettant par exemple un passage automatisé de codes matlab vers une implémentation binaire pour Processeurs et DSP !*

Les besoins de vérification dans les conceptions actuelles

Réutilisation (reuse) et vérification



- SOC : Ex.: circuit de 10 M.portes
  - 6000 T/pers/mois => 140 personnes pendant une année...!!!
    - Multiples fonctions ⇔ Multiples compétences,
    - Une seule (grosse) équipe ? Augmente le coût du projet, difficile à manager efficacement, difficulté de communication (⇒ mauvaises compréhensions,, bugs),
  - Idée pour réduire le time-to-market : Utilisation de blocs préconçus (en interne ou hors de l'entreprise) pour certaines fonctions du système,
- Concept de réutilisation ↔ IP, composants virtuels, blocs
  - Réutilisables,...
  - À l'origine : pour le HW, mais existe aussi pour le SW

Réutilisation (reuse) et vérification

Niveau	Format	Flot de réutilisation	Flexibilité
Soft	Description VHDL/Verilog RTL	Synthèse RTL	**Architecture modulable** Indépendante de la technologie
Firm	Netlist (EDIF)	Placement & Routage	**Architecture figée** Optimisée pour une technologie
Hard	Masques (GDSII)	Intégration système	**Architecture figée** Placée/routée pour une technologie

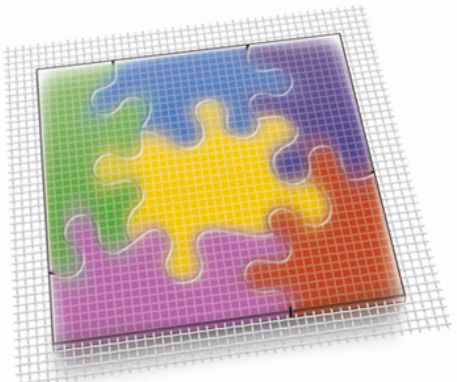
- Données nécessaires à l'intégrateur :
  - Modèle simulable
  - Validation fonctionnelle (méthode(s) utilisée(s) + testbenches)
  - Performances temporelles, surface, consommation
  - Modèle synthétisables + scripts (contraintes,…)
  - …

Le besoin de confiance lors de la réutilisation

- La réutilisation, une question de confiance (Reuse is about Trust)
  - L'enjeu de la réutilisation est crucial,
  - Les concepteurs ont peu de temps pour intégrer dans leurs systèmes des IPs (fournis par des personnes différentes),
  - Nécessaire de vérifier que le bloc réutilisé a été méticuleusement vérifié en accord avec les spécifications.
  - Les vérifications fonctionnelles servent à prouver la qualité du circuit.
- Diminution très rapide de la fiabilité d'un assemblage de blocs (0,99n)

Ensemble de composants pré-existants

Sélection, assemblage et mise au point



Circuit = Agglomérat de composants développés sur mesure et/ou pré-existants

Réutilisation (reuse) et vérification

- Mais concept basé sur la confiance (qui est responsable en cas de non/mal fonctionnement avec le reste du système ?)
- Pour renforcer la confiance (…et donc faciliter les ventes), il est important de pouvoir démontrer que le composant réutilisable a été méticuleusement vérifié en accord avec les spécifications.
  - ➡ La vérification fonctionnelle sert à démontrer la qualité du composant

Qualité/fiabilité d'un design : probabilité de bon fonctionnement (1st-time working silicon)

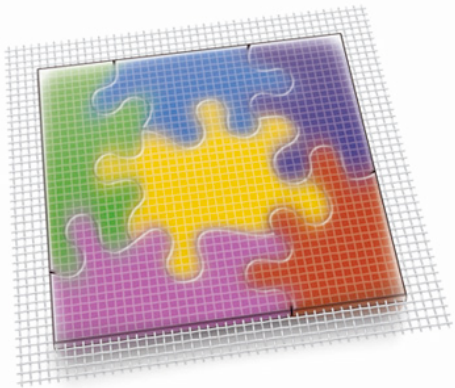
- ASIC de 150 k portes : **0,9** (circuit seul) (NB : fonctionnement correct dans un système : 0,5 (tout n'est jamais prévu...))
- SoC de 1,5 M portes (SoC peu complexe…)
  - ✓ Circuit décomposé en 10 parties de 150 k portes développées spécifiquement pour le projet
    $(0,9)^{10}$ = **0,35**
  - ✓ Circuit constitué de 10 IP de 150 k portes (parties pré-conçues et pré-vérifiées)
    $(0,99)^{10}$ = **0,9**
  - ✓ Circuit constitué de 8 IP et 2 parties (150 k portes) développées spécifiquement pour le projet
    (0,99)8. (0,9)2 = 0,75

Les composants virtuels dédiés à la vérification

- Généralement non synthétisables car ils sont disponible afin de vérifier qu'acheter un bloc permet de répondre à un besoin,
- Fournis à haut niveau d'abstraction mais possédant un comportement au cycle prés,
  - ➡ Vitesse de simulation élevée,
  - ➡ Quelle est l'adéquation avec le circuit physique ?
  - ➡ Tous les problèmes sont ils repérables ?

Ensemble de composants pré-existants

Sélection et assemblage

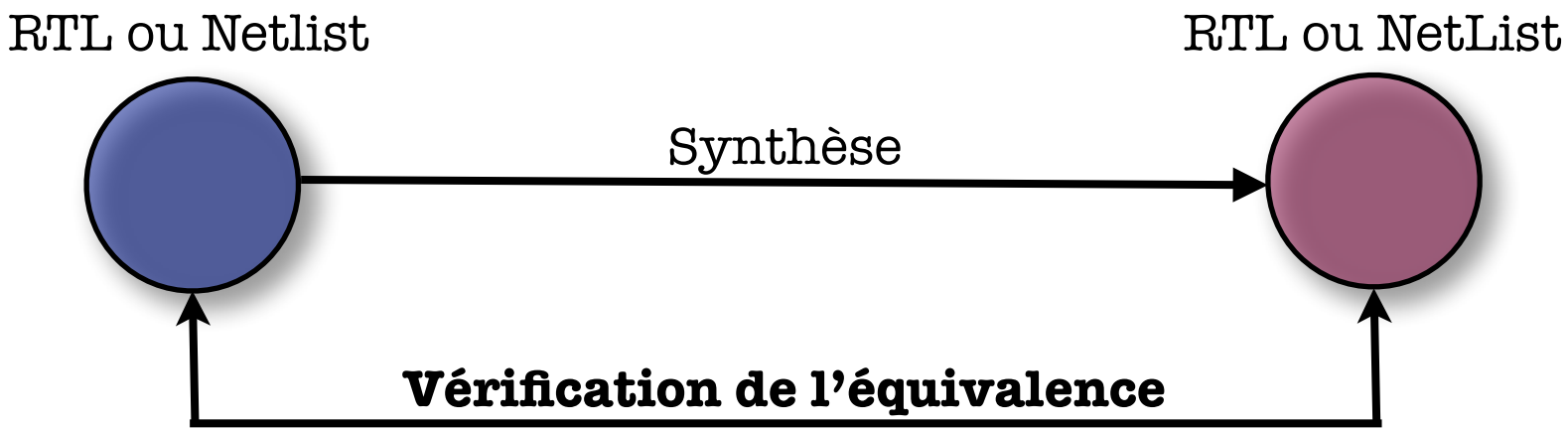


Systèmes complexes

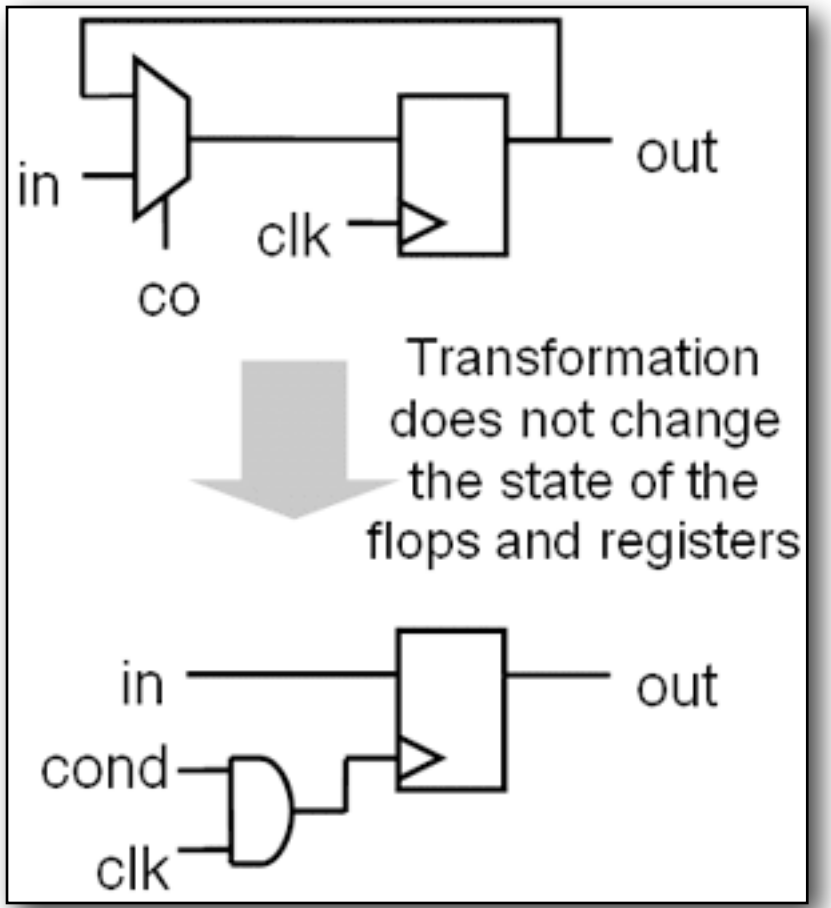
Les outils de preuve formelle (Equivalence / Model checking)

Vérification formelle - Equivalence Checking

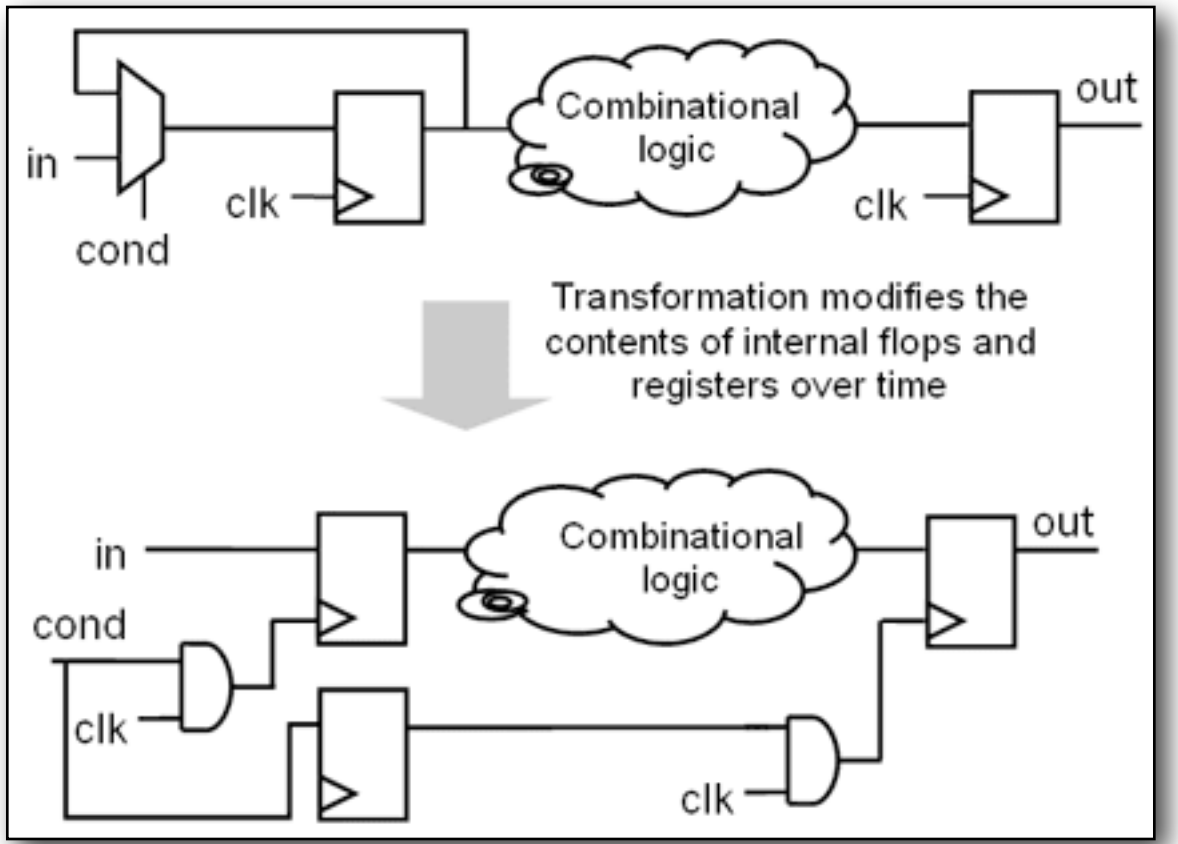
- Équivalence logique prouvée mathématiquement (fonctions booléennes et séquentielles)
- Méthode généralement employée afin de comparer 2 netlists (Exemple : Netlist d'origine <=> Netlist avec Scan-Based, …)
  - ➡ Vérification de l'équivalence logique des différents designs de niveau RTL
  - ➡ Vérification de l'équivalence logique entre modèle RTL et les modèle de niveau portes (vérification non nécessaire si l'on a une confiance aveugle dans l'outil de synthèse…)



Equivalence checking - Exemples (1)

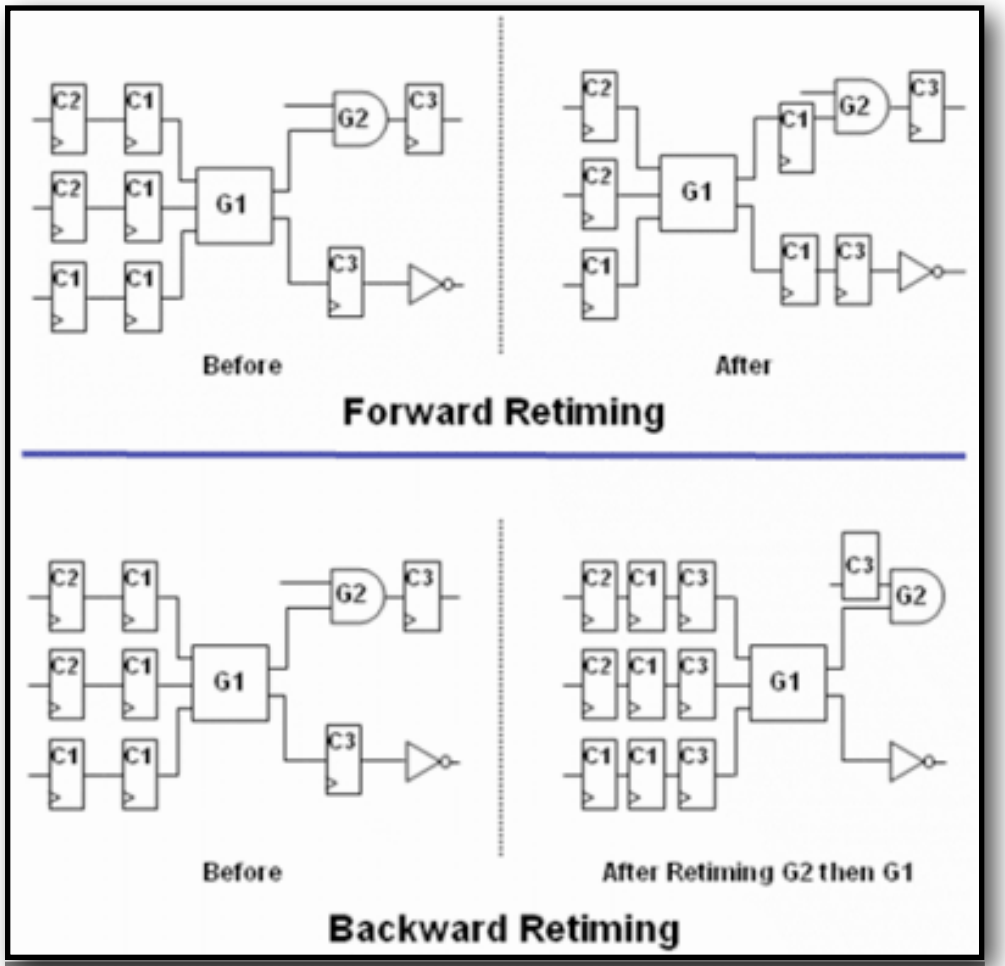


Modification architecturale impliquant une modification des équations logiques mais sans changer la fonctionnalité.

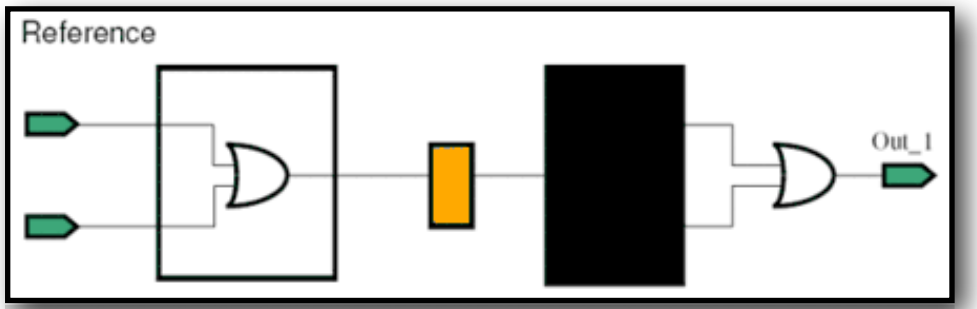


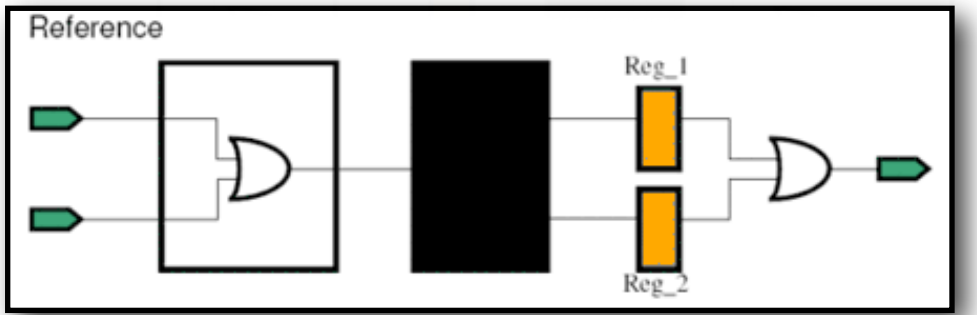
Modification architecturale modifiant les équations logiques et les fonctionnement du système sans altérer les résultats.

Equivalence checking - Exemples (2)



Les différents designs sont ils équivalents les uns par rapport aux autres ?





Equivalence checking - Exemples (3)

Idempotent

$A+A = A$ $\qquad A \cdot A = A$

Associative

$(A+B)+C = A+(B+C)$ $\qquad (A \cdot B) \cdot C = A \cdot (B \cdot C)$

Commutative

$A+B = B+A$ $\qquad A \cdot B = B \cdot A$

Distributive

$A+(B \cdot C) = (A+B) \cdot (A+C)$ $\qquad A \cdot (B+C) = (A \cdot B)+(A \cdot C)$

Identity

$A+0 = A$ $\qquad A+1 = 1$

$A \cdot 0 = 0$ $\qquad A \cdot 1 = A$

Complement

$A+\bar{A} = 1$ $\qquad \overline{(\bar{A})} = A$

$A \cdot \bar{A} = 0$ $\qquad \bar{1} = 0$

DeMorgan's

$\overline{(A+B)} = \bar{A} \cdot \bar{B}$ $\qquad \overline{(A \cdot B)} = \bar{A}+\bar{B}$

Duality

interchange AND and OR operators, as well as all Universal, and Null sets. The resulting equation is equivalent to the original.

Note: When simplifying Boolean algebra, OR operators have a lower priority, so they should be manipulated first. NOT operators have the highest priority, so they should be simplified last. Consider the example from before.

$X = \overline{(A+B \cdot C)}+A \cdot (B+\bar{C})$

$X = \overline{(A)}+\overline{(B \cdot C)}+A \cdot (B+\bar{C})$ ← The higher priority operators are put in parenthases

$X = \overline{(A)} \cdot \overline{(B \cdot C)}+A \cdot (B+\bar{C})$ ← DeMorgan's theorem is applied

$X = \bar{A} \cdot (\bar{B}+\bar{C})+A \cdot (B+\bar{C})$ ← DeMorgan's theorem is applied again

$X = \bar{A} \cdot \bar{B}+\bar{A} \cdot \bar{C}+A \cdot B+A \cdot \bar{C}$ ← The equation is expanded

$X = \bar{A} \cdot \bar{B}+(\bar{A} \cdot \bar{C}+A \cdot \bar{C})+A \cdot B$ ← Terms with common terms are collected, here it is only NOT C

$X = \bar{A} \cdot \bar{B}+\bar{C} \cdot (\bar{A}+A)+A \cdot B$ ← The redundant term is eliminated

$X = \bar{A} \cdot \bar{B}+\bar{C}+A \cdot B$ ← A Boolean axiom is applied to simplify the equation further

$A = \bar{B} \cdot (C \cdot (\bar{D}+E+C)+\bar{F} \cdot C)$

$A = \bar{B} \cdot (\bar{D} \cdot C+E \cdot C+C \cdot C+\bar{F} \cdot C)$ (1)

$A = \bar{B} \cdot (\bar{D} \cdot C+E \cdot C+C+\bar{F} \cdot C)$ (2)

$A = \bar{B} \cdot C \cdot (\bar{D}+E+1+\bar{F})$ (3)

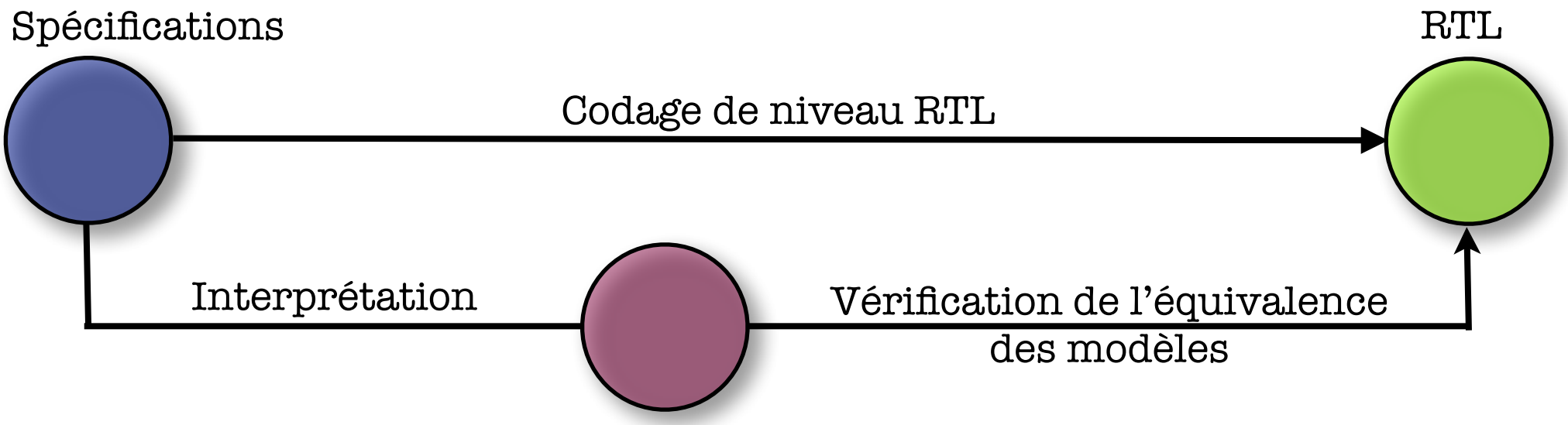
$A = \bar{B} \cdot C \cdot (1)$ (4)

$A = \bar{B} \cdot C$ (5)

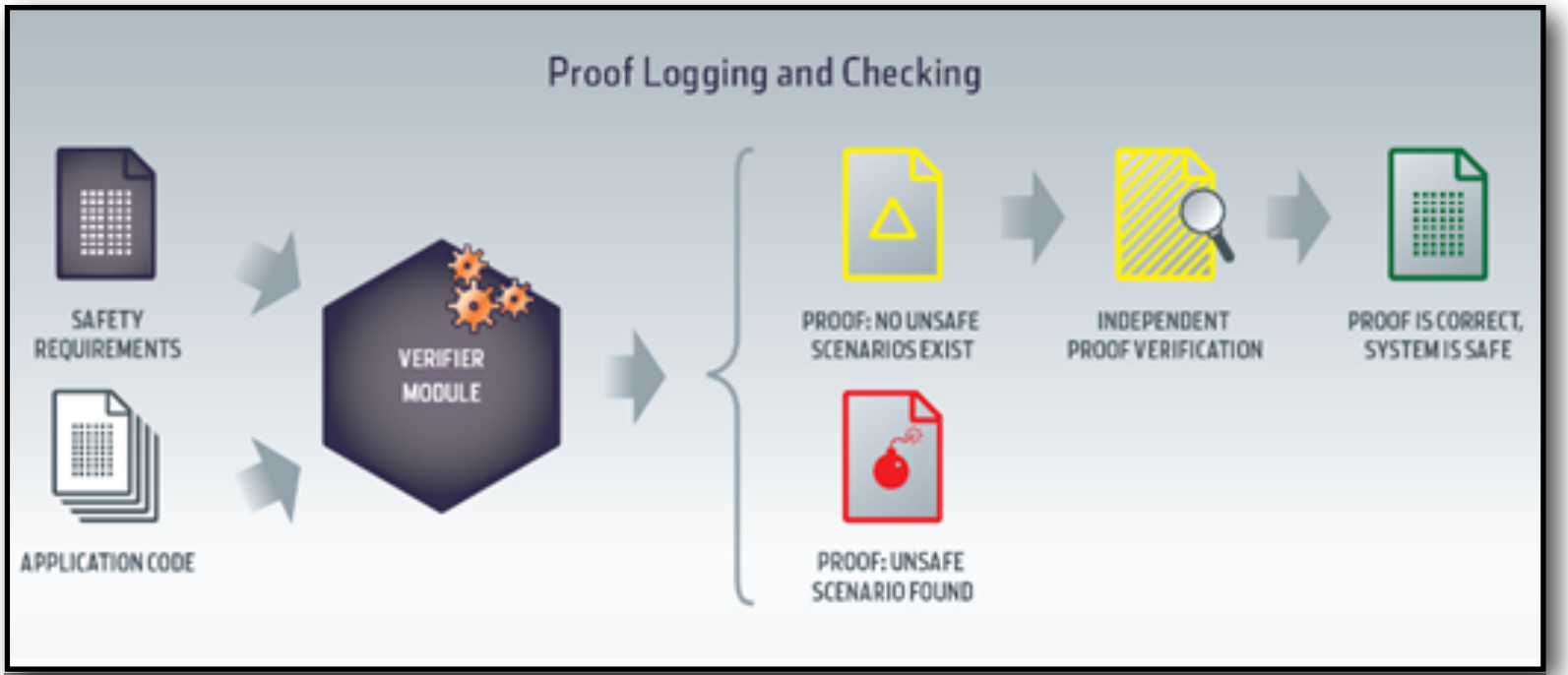
http://www.eod.gvsu.edu/~jackh/books/plcs/html/plcs.html

Vérification formelle - Model Checking

- On prouve formellement que les caractéristiques (comportement) de la spécification sont respectées dans le design RTL
  - Définition des propriétés qui doivent être prouvées (ex.: pas d'état isolé (impossible à atteindre) dans la machine d'état du composant, vérification de l'interface : réponse (signal acknoledge) suite à une requête, …)
  - Difficulté actuellement pour exprimer correctement les propriétés à vérifier. Utilisation nécessaire d'un langage formel



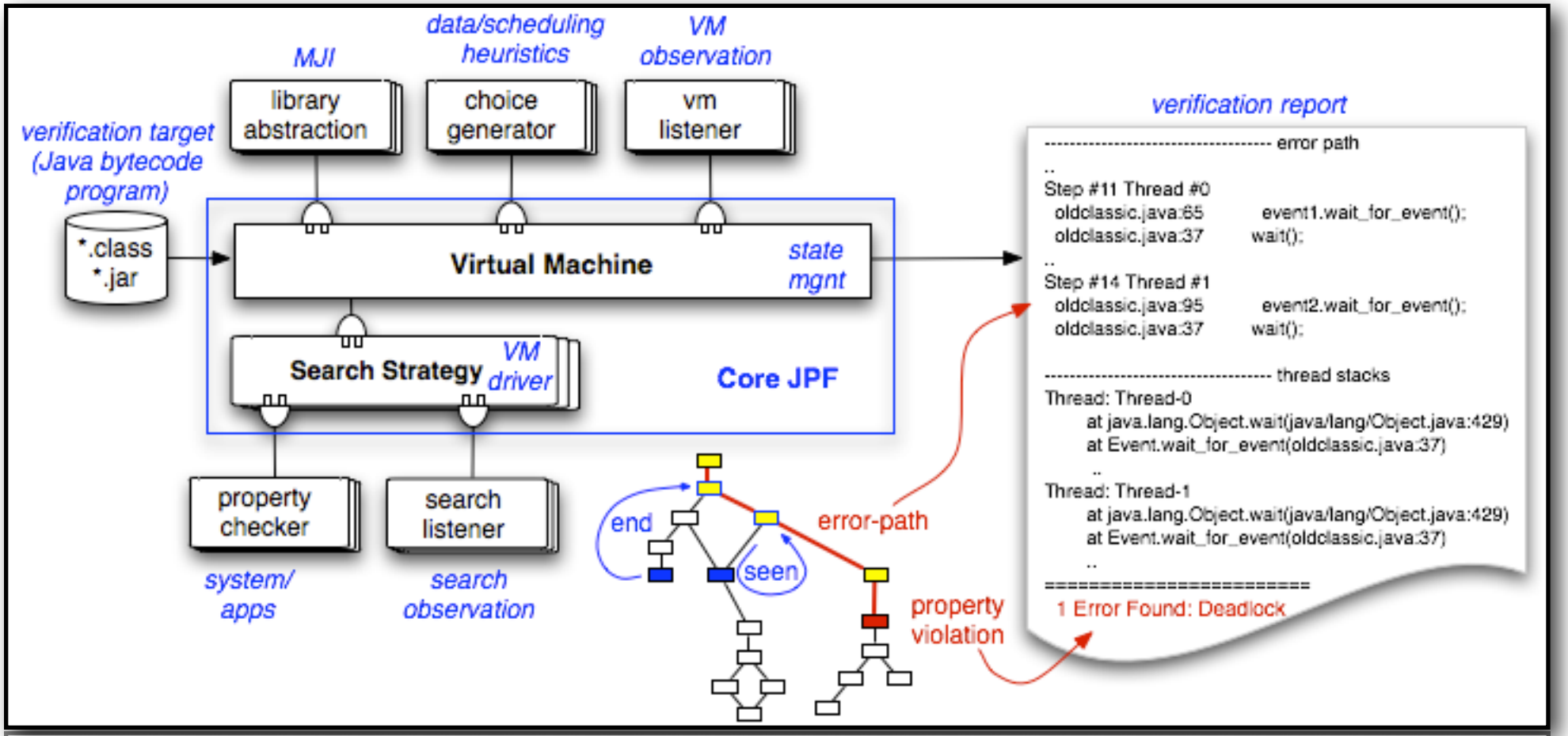
Model Checking - Outil iLock



Les outils de comparaison de modèles (Model Checking) se base sur l'expression du modèle de solution ainsi que sur une description des règles que ce dernier doit respecter.

http://www.prover.com/products/ilock/miniwp.xml

Model Checking - Outil JavaPathFinder

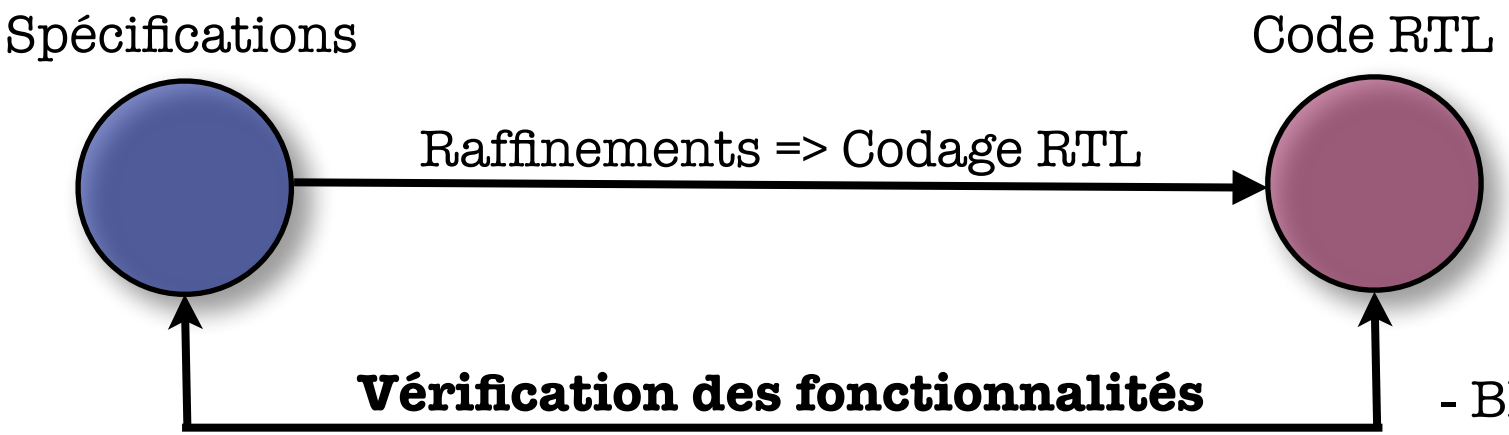


http://javapathfinder.sourceforge.net/

Les approches employées pour faire de la vérification fonctionnelle

La vérification fonctionnelle

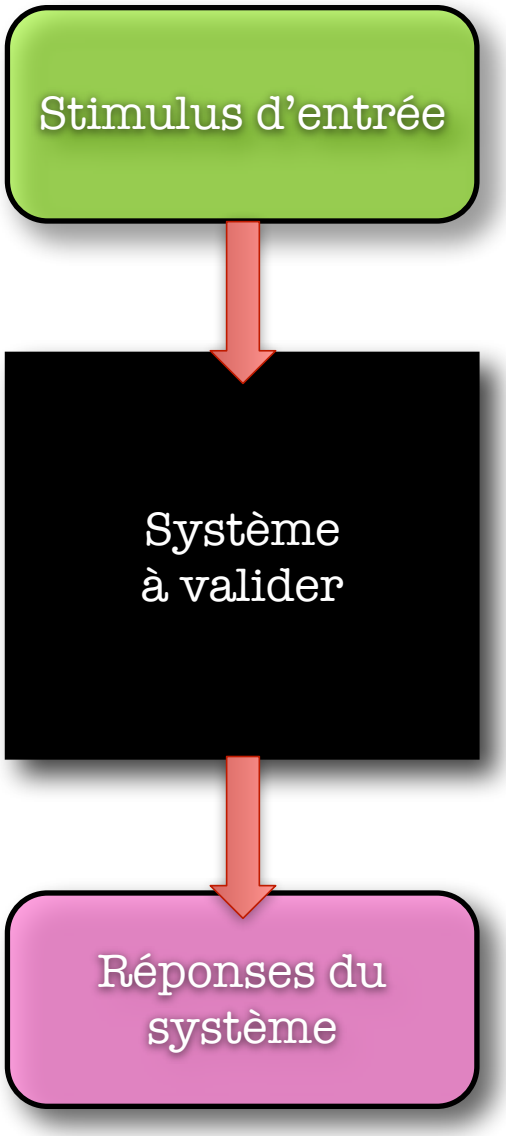
- Introduction à la vérification fonctionnelle (Functionnal verification)
  - L'objectif est de vérifier que le design implémente de manière correcte les différentes fonctionnalités souhaitées (vérifier, mais pas prouver…),
  - Permet de vérifier qu'il n'y a pas eu d'erreur d'interprétation de la spécification et que les transformations n'ont pas engendré d'erreur.



- Black-box verification
- White-box verification
- Grey-box verification

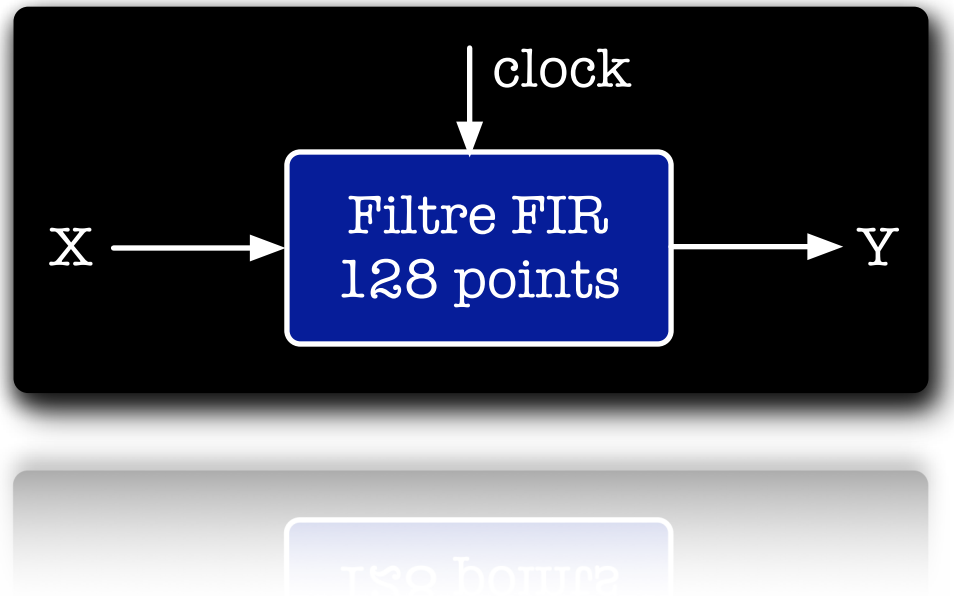
Vérification fonctionnelle - Black box verification

- Avantages
  - Approche basée sur la non connaissance de l'implémentation,
  - Utilisation exclusive des interfaces disponibles pour vérifier le composant,
  - Efficace dans les étapes de raffinement ,
- Inconvénients
  - Impossible de forcer des états internes (pour gagner du temps par ex.),
  - Difficile d'observer isolément certaines fonctionnalités,
  - Si erreur, difficile de localiser la source du problème,



Exemple de mise en oeuvre

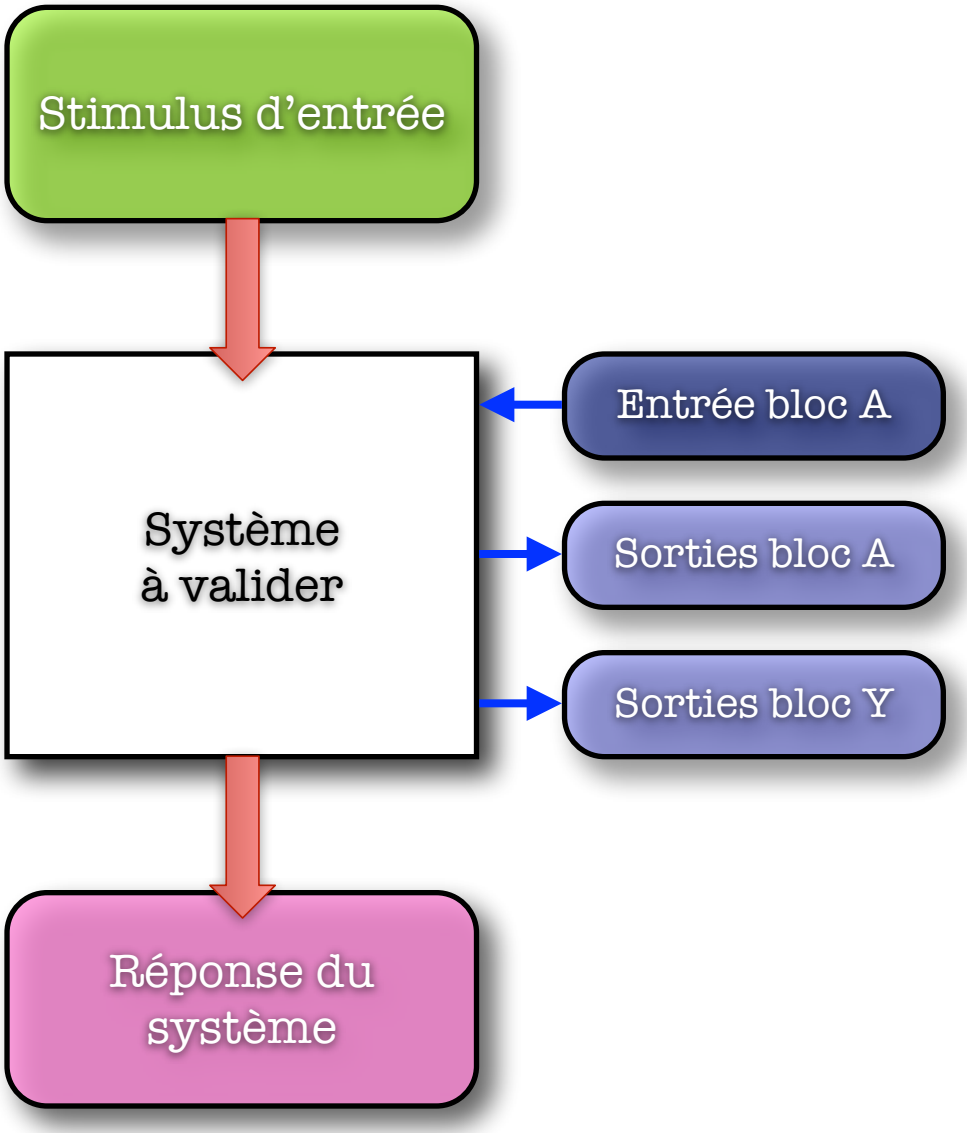
$$y = \sum_{i=0}^{127} X(i) \times H(127 - i)$$



Dans le cadre du test de ce composant on observe uniquement la sortie en fonction des valeurs d'entrée que l'on impose.

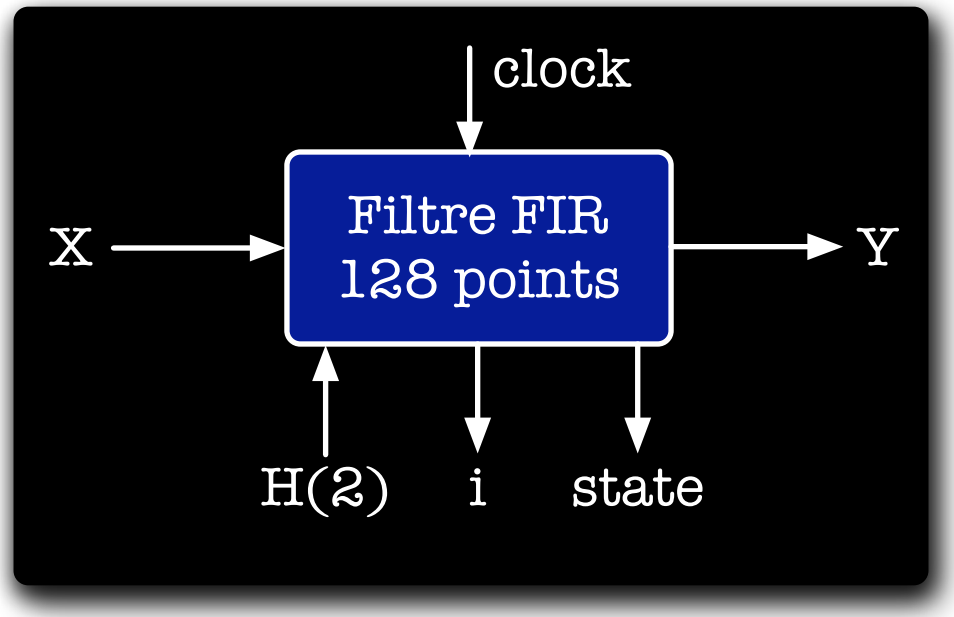
Vérification fonctionnelle - White box verification

- Visibilité et contrôle total de la structure interne du composant
  - Connaissance nécessaire de l'implémentation
  - Implique généralement de modifier le circuit afin de pouvoir le vérifier,
  - Liée à une implémentation spécifique (si modification du design, modification probable du testbench)
- Réservée principalement au niveau de vérification système,
  - Eviter les simulations longues
  - système composé de blocs indépendants interconnectés.



Exemple de mise en oeuvre

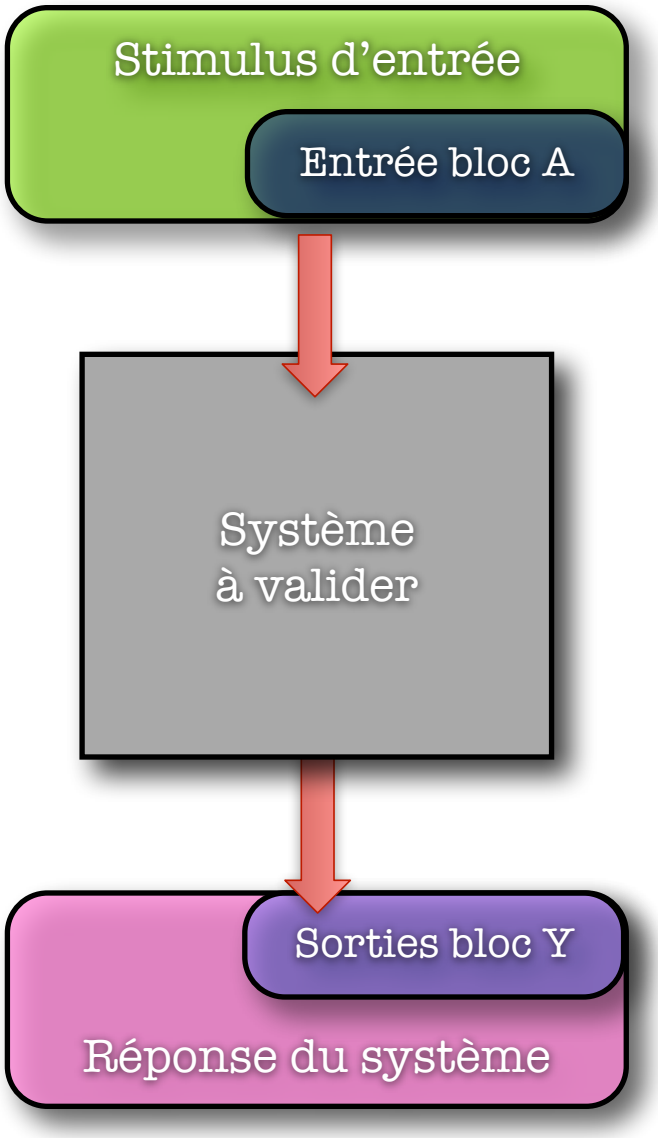
$$y = \sum_{i=0}^{127} X(i) \times H(127 - i)$$



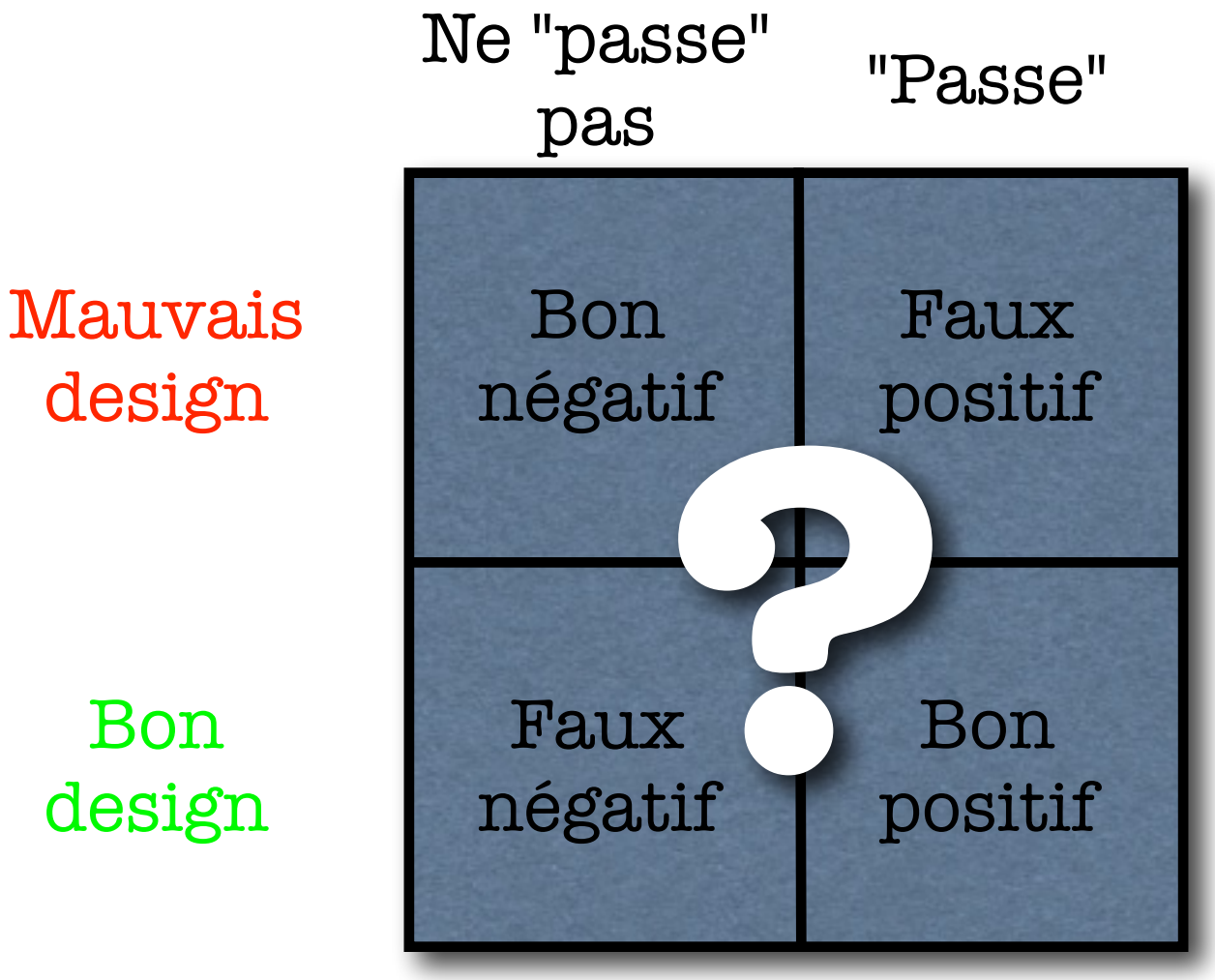
Dans le cadre du test de ce composant on observe la sortie ainsi que des données temporaires et l'on force la valeur de l'entrée ainsi que des valeurs internes.

Vérification fonctionnelle - Grey box verification

- C'est un compromis entre les approches,
  - ➡ Black-box (trop globale),
  - ➡ White-box (trop proche de l'implémentation, non portable)
- Ajout de modifications non fonctionnelles pour améliorer l'observatibilité,
  - ➡ Exemple.: un registre accessible via les E/S pour vérifier son état, pour forcer un état, etc.
- Contrôle et observation au niveau interfaces (comme black-box)

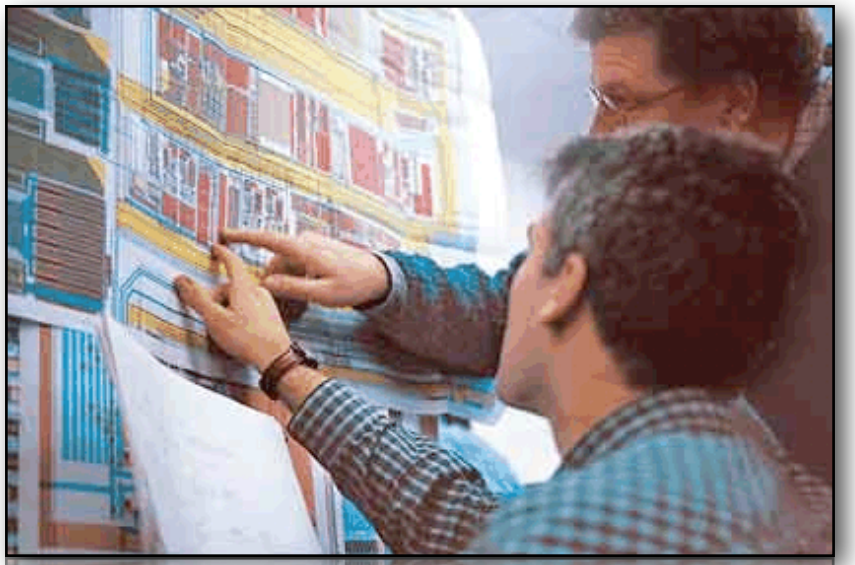


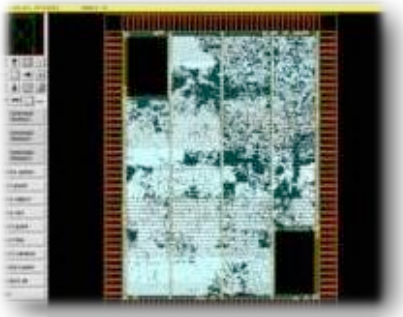
L'utilisation de vecteurs de tests est il simple ?

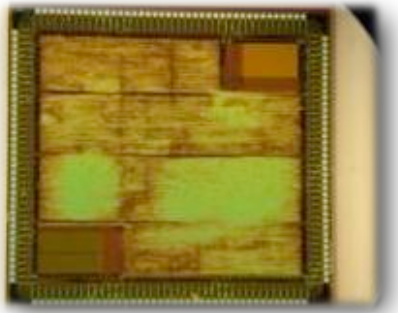


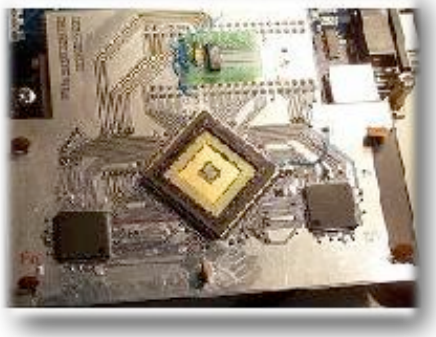
Concevoir pour vérifier (Design for testability)

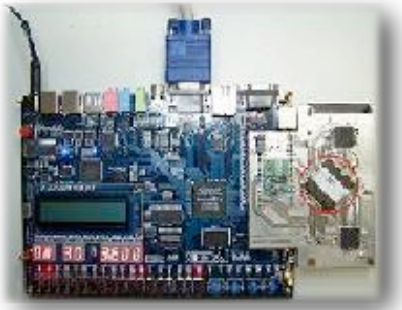
- Conception adaptée depuis longtemps pour faciliter la vérification,
  - ➡ Pourquoi ce circuit ne fonctionne-t-il pas ?
  - ➡ Où se situe le problème physique ?
  - ➡ Exemple : Scan-Based (mise en série des registres en mode Scan),
    - Testabilité améliorée,
    - Surcoût en surface & consommation.











Prendre en considération la vérification lors de la conception

- Concevoir pour vérifier (Design for verification)
  - Il est nécessaire d'adapter la conception d'un circuit pour faciliter sa vérification fonctionnelle (légitime car la vérification est souvent plus complexe que le développement lui-même),
  - La vérification du système doit être prise en considération dès la phase de spécification et non uniquement durant le développement,
  - Le concepteur doit être en mesure de répondre à :
    - Qu'est-ce que cela est supposé faire ?
    - Mais aussi : Comment va-t-on faire pour vérifier cette fonctionnalité ?
  - Qq. techniques favorables au "design for verification" :
    - Interfaces bien définies
    - Séparation claire des différentes fonctions en unités les plus indépendantes possibles
    - Disposer de registres accessibles via les E/S (grey-box verification)
    - Disposer de mux/demux pour isoler/contourner certaines unités fonctionnelles

Vérifier, certainement mais dans quel but ?

- La vérification pour la réutilisation
  - Besoin de vérifier de manière approfondie un design (qui va et sera réutilisé) de manière équivalente à un produit destiné au client,
  - Les designs réutilisables sont partiellement génériques, impliquant la nécessité de vérifier fonctionnellement toutes les configurations possibles,
- La réutilisation de la vérification
  - Il est possible de réutiliser des designs, ce qui implique qu'il est aussi possible de réutiliser tout ou partie des tests fonctionnels,
  - Pas toujours possible de réexploiter entièrement un banc de test réalisé pour une autre application (modes et contraintes de fonctionnement différents),
  - Le changement de niveau d'abstraction peut impliquer une réécriture des bancs de tests partielle ou totale. (changement de langage, typage des données float <=> std_logic_vector, etc.).

Le coût de la vérification dans la réutilisation

- La vérification est un mal nécessaire
  - Elle coûte beaucoup trop cher,
  - Elle dure évidement trop longtemps,
  - Elle n'est pas à proprement parlé génératrice d'argent,
- La vérification est un processus sans fin
  - Vous pouvez prouver la présence d'erreurs, mais pas leur absence,
  - Estimations statistiques possibles du nombre d'erreurs restantes,
- L'erreur que l'on va peut être trouver est elle assez sévère pour investir du temps et de l'argent pour la corriger ?
- La vérification est un processus à part entière et ne peut se résumer à l'utilisation de testbenchs.

Les différents type de testbench

Architecture typique d'un testbench

- La conversion des données est réalisée dans les interfaces de communication,
  - ➡ Evolutivité des bancs de tests au cours de la conception (lors du processus de raffinement, seules les interfaces changent),
- Diminuer l'intervention humaine autant que possible (gain de temps, limitation du risque d'erreurs),
- Génération des données
  - ➡ A la main ? Utilisation d'outils ou de scripts ?
- Analyse des réponses
  - ➡ Viser une vérification automatique signalant que les tests ont été passés avec succès ou qu'ils ont échoués,
- Le fait de retirer le harnais de tests ne compromet pas la validation fonctionnelle opérée !

Cette approche est vraie pour l'ensemble des langages Hardware & Software

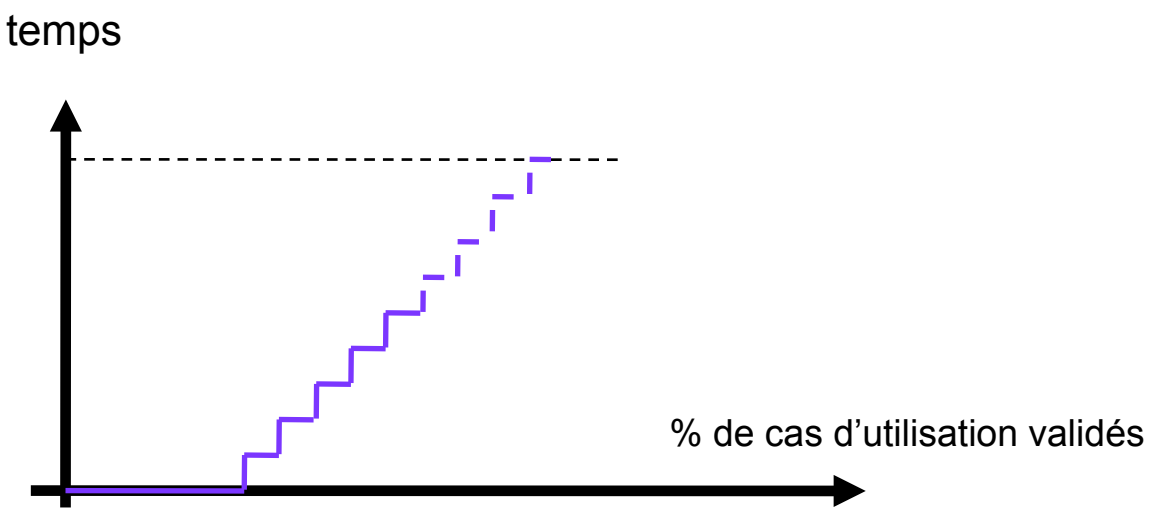
Génération : données générées à façon *

- Time is money !
  - Toujours essayer de minimiser le nombre de tests à valider (sans rien oublier…).
  - Vérifier seulement ce qui est nécessaire (attention à la réutilisation)…
- Grouper les tests en cas d'utilisation
  - Propriétés à vérifier qui demandent la même configuration, la même granularité, la même stratégie de vérification
  - Assigner à une même personne chaque cas d'utilisation
- Passer des cas d'utilisation aux testbenches
  - Lier les testbenches aux cas d'utilisation
  - Assigner à des personnes différentes la réalisation des différents bancs de test

* données générées à façon : générées à la main (codées en dur)

Génération : approche données générées à façon

- Vérifier les bancs de tests,
  - ➡ Comment s'assurer de la validité des testbenches écrits (données d'entrées) ?
    - Faire vérifier/utiliser ces derniers par d'autres personnes (redondance) (implique testbenches bien structurés, commentés, …),
- L'intérêt des cas d'utilisation est de fournir une mesure de l'avancement du projet => combien de cas sont fonctionnels ?



La génération automatisée de valeurs

- Génération de stimulus aléatoires
  - Un avantage de la génération est un gain de temps versus un codage manuel,
  - Autre avantage, une génération de configurations non imaginées par l'équipe de vérification (+ objectif que l'être humain…),
- Génération aléatoire ne veut pas dire générer n'importe quoi (la génération de valeurs aléatoires n'est pas difficile)
  - Vérifier seulement ce qui est nécessaire => diriger les testbenches selon les propriétés à vérifier (s'assurer de la validité des stimulus générés),
  - Très utile : functional coverage…
  - Difficulté : écriture du générateur aléatoire (contraint) (ex. séquence particulière de données).

Ecriture d'un générateur aléatoire contraint

- Exemple d'un générateur aléatoire pour un composant de division sur 32 bits,
- Caractéristiques d'utilisation :
  - Le dividende est toujours positif ou nul avec pour valeur maximale 65537,
  - Le diviseur est positif ou négatif mais jamais nul avec un espace de variation compris dans [-129, 2048],
- Ecrivez un générateur,
  - pour la fonction nommé MaDivision(int, int) sans contraindre le générateur,
  - en contraignant le générateur afin de limiter l'étendu des tests,

```
int MaDivision(int A, int B){
      int C = 0;
      bool signe = (B>=0)?0:1;
      B = abs(B);
      int cpt = 0;
      while( true ){
            if(A >= B){
                  C = C + 0x1;
                  A = A - B;
            }else{
                  return (signe==0)?(C):(-C);
            }
            if(cpt++ == 70000)
                  return -70000;
      }
}
```

Exemple de code algorithmique permettant de réaliser une division par soustractions.

Ecriture d'un générateur aléatoire contraint

```
int Generateur_1( ){
     unsigned int limite = 10000000;
     unsigned int erreur = 0;
     while( (limite--) != 0 ){
           int a = rand() - RAND_MAX/2;
           int b = rand() - RAND_MAX/2;
           if( (a/b) != MaDivision(a,b) ){
                 erreur += 1;
           }
     }
     return erreur/100000;
}
```

Ce premier testbench permet de tester tous les cas possibles de division (int/int). Malheureusement, il va planter à l'exécution et fournir des faux positifs !

```
bool Generateur_2( ){
     unsigned int limite = 10000000;
     unsigned int erreur = 0;
     while( (limite--) != 0 ){

           /* On recadre "B" dans l'intervalle */
           int a = abs( rand() - (RAND_MAX/2) ) % 65539;

           /* On recadre "A" dans l'intervalle */
           int b = rand();
           if( b < -129 ) b /= (RAND_MAX/2/129);
           if( b > 2048 ) b /= (RAND_MAX/2/2048);
           if( b == 0   ) b = 1;

           if( (a/b) != MaDivision(a,b) ){
                 erreur += 1;
           }
     }
     return erreur/100000;
}
```

Ce second testbench est plus complexe à développer mais il ne fourni que des résultats intéressants vis a vis de la fonction a tester...

Self-checking testbenches

- La vérification manuelle des réponses n'est pas une solution acceptable à long terme (temps passé, risque d'erreurs)
  - ➡ Il est donc nécessaire de déployer des bancs de test qui vérifient les résultats de manière automatisée,
  - ➡ Le développement de scripts / code prend du temps mais permet d'en gagner beaucoup plus au final !
- Différentes techniques possibles
  - ➡ Réponses codées en dur (repose donc sur le principe que les stimulus d'entrée sont connus (codés en dur). Testbenches aléatoires exclus…)
  - ➡ Marquage de données : réservé aux cas d'application où une partie des données d'entrée est répétée en sortie (typiquement transfert de données, ex. routeur réseau, …). La partie répétée en sortie est "aléatoire" (génération aléatoire utilisée pour la partie correspondante dans les stimulus d'entrée), la partie réellement transformée est codée en dur (comme pour l'entrée)
  - ➡ Utilisation d'un modèle de référence afin de valider le comportement.

Les tests unitaires - Exemple en langage C

Cette technique est aussi utilisée dans le cadre des tests de non régression.

Exemple de fonctionnalité à tester (composant logiciel ou matériel)

```
int mult(int a, int b){
    int c = 0;
    bool neg = (b>=0)?false:true;
    b = (b>=0)?b:-b;
    while( b != 0 ){
        c = c + a;
        b = b - 1;
    }
    return (neg)?c:-c;
}
```

Programme développé uniquement afin de valider le fonctionnement de la méthode "mult"

```
int main( ){
    assert( mult(2,  3) ==  6 );
    assert( mult(-2,-3) ==  6 );
    assert( mult(2, -3) == -6 );
    assert( mult(-2, 3) == -6 );
    assert( mult(2,  0) ==  0 );
    // ... ... ... ... ...

    return 1;
}
```

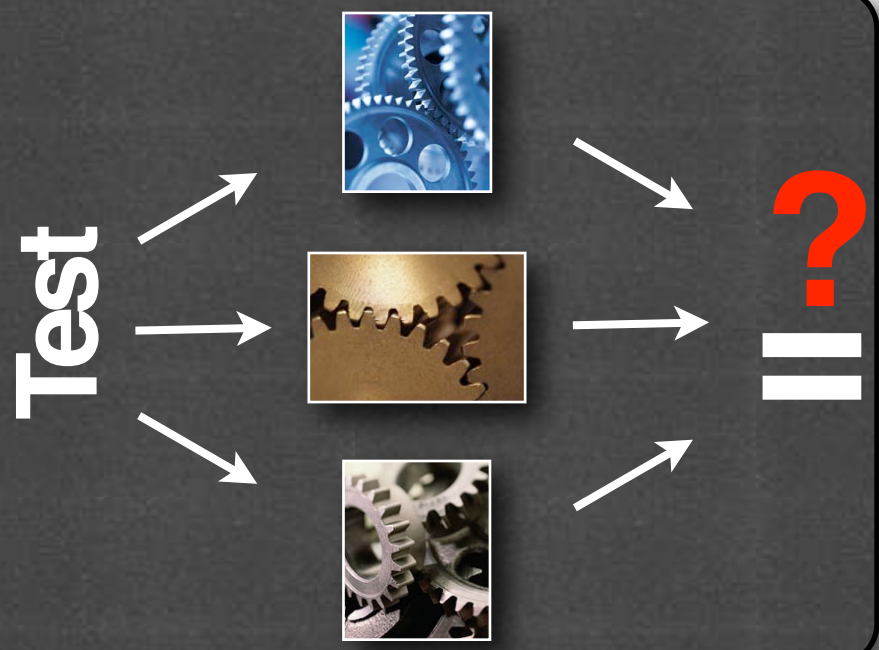
Les pistes proposées afin d'améliorer les processus de vérifications

Travailler sur la réduction des risques...

Comment réduire les risques d'erreurs lors des phases de conception et de test ?

- Automatisation
- Poka-Yoke
- Redondance

Chaque transformation est dupliquée et vérifiée indépendamment (n fois)



Méthode qui est à la fois, la plus basique et la plus coûteuse…

Réservée aux domaines nécessitant de très hauts taux de fiabilité (aérospatiale)

Une approche simple et efficace afin d'améliorer la vérification

- Organisation de la conception et de la vérification…
  - ➡ Dissocier qui conçoit (designer) de qui vérifie
    - Objectivité, pas d'a priori pour vérifier tout et n'importe quoi,
    - Meilleure solution pour trouver des erreurs,
    - A chacun son métier et ses outils / techniques,

Les outils d'aide à la vérification (analyse du code source)

Les « Linting tools »

- « lint » est un outil du monde UNIX qui :
  - Analysait un programme écrit en C
  - Rapportait un certain nombre d'interrogations "légitimes"
  - Identifiait les problèmes syntaxiques et sémantiques possibles
- « lint » tentait d'identifier les erreurs les plus fréquemment écrites par les concepteurs de logiciels, de manière rapide et efficace (avant l'exécution du programme)

Exemple de détection de problèmes

```
int function( int a, int b )
    return f(a,b);
}

int main( void ){
    int a, b=0;
    long d = 32758;
    char c = function(a, b);
    if( c > d ){
        return 1;
    } else {
        return 0;
    }
    return c;
}
```

Quels problèmes potentiels un outil pourrait-il percevoir dans cet exemple ?

« a » non initialisé

Transtypage implicite avec perte d'informations

Condition toujours fausse

Code mort (jamais exécuté)

Code mort (jamais exécuté)

Conclusion sur les «linting tools»

- Avantages
  - Analyse rapide du code source,
  - Analyse statique : pas besoin de stimulus (contrairement aux autres méthodes/outils),
  - Permettent de gagner du temps (la détection et la résolution des problèmes à l'exécution sont plus complexes),
  - Performants pour détecter les erreurs "statiques" usuelles,
- Inconvénients
  - Risque de faux négatifs (erreur détectée alors que le design est correct) important, impliquant une perte de temps et/ou une réceptivité moins importante aux messages,
    - Utiliser un bon standard de codage peut aider à éviter les faux négatifs,
  - Détection de certains types de problèmes uniquement (problèmes trouvés de manière statique à partir de l'algorithme),
  - Exemple : s = a + b : si a est écrit(tapé) à la place de c => erreur non détectée
    - Risque de faux positifs (erreurs non détectées)
- Bilan => les Linting tools sont utiles mais insuffisants !

Detection des erreurs usuelles en VHDL

- VHDL => Langage fortement typé
  - La plupart des analyses propres à un linting tool sont actellement faites par le compilateur,
  - Exemple de problème classique pas détecté à la compilation : avec le type std_logic (type résolu de std_ulogic) : un même signal peut avoir plusieurs drivers (car type résolu). Or seul cas où cela est vraiment nécessaire : les bus.

```
ARCHITECTURE exemple OF ...
    SIGNAL s1: std_logic;
    SIGNAL sl: std_logic;

    PROCESS( donnée )
    BEGIN
        s1 <= donnée_de_type_std_logic;
        sl <= not donnée_de_type_std_logic;
    END process;
END exemple;
```

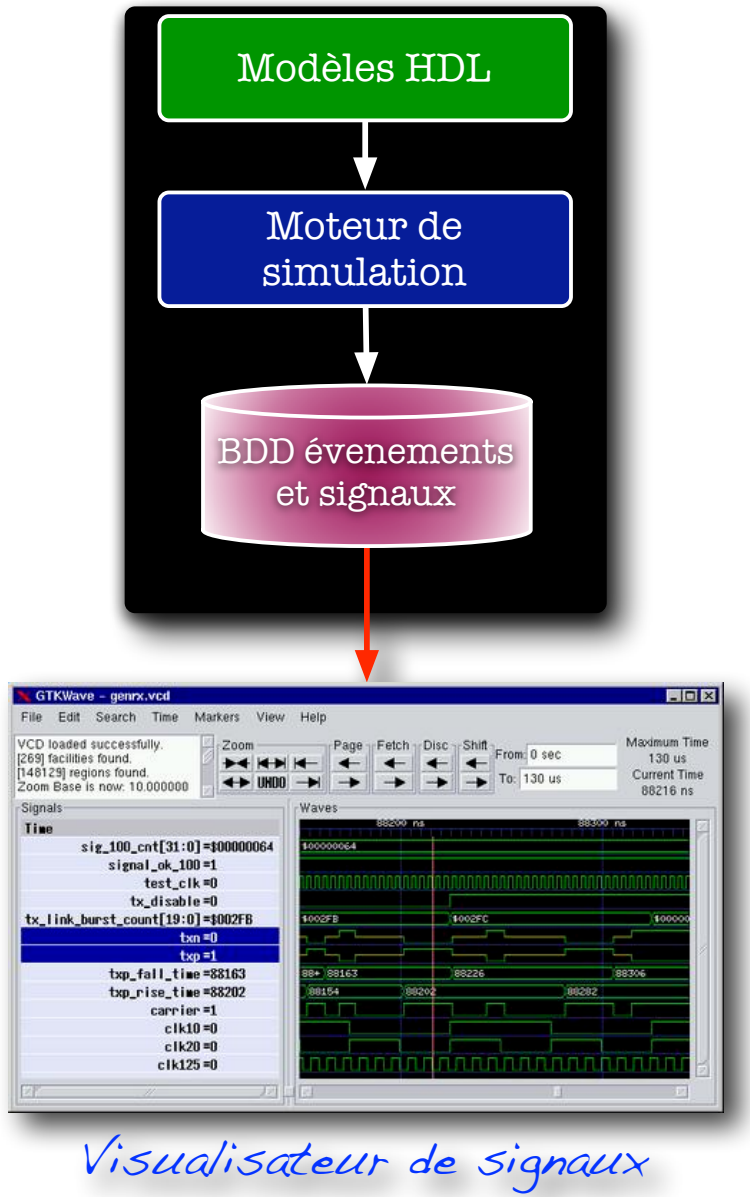
2 drivers pour s1... alors qu'il s'agissait d'une faute de frappe (s1 <=> sl). Ne serait pas passé à la compilation avec le type non résolu std_ulogic

- Verilog
  - Linting tools plus utiles qu'en VHDL ((Verilog-95), un peu moins avec Verilog-2001))

Les outils d'aide à la vérification (analyse de la simulation fonctionnelle)

Visualisateurs de signaux (Waveform viewers)

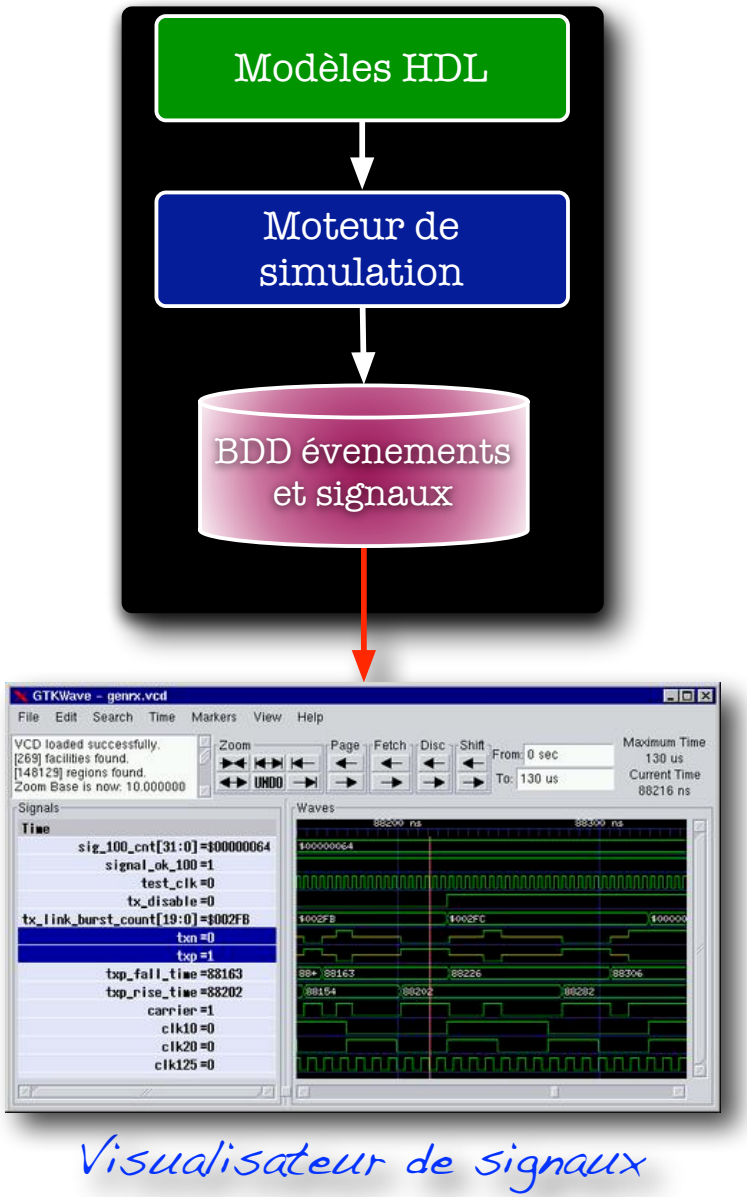
- Les outils de visualisation sont utilisés conjointement avec les simulateurs,
- Affichent les changements de valeurs (chronogrammes).
  - ➡ Indispensables durant la phase de développement d'un composant et des testbenches
- Analyse à postériori au travers du fichier de trace,
  - ➡ L'enregistrement d'une trace réduit les performances d'un simulateur,
  - ➡ N'enregistrer que ce qui est nécessaire.



Visualisateur de signaux

Visualisateurs de signaux (Waveform viewers)

- Les visualisateurs de signaux sont des outils inadaptés à la validation (vérif. fonctionnelle)
  - ➡ Complexité (analyse de centaines de signaux sur des durées longues),
- Certains outils permettent de comparer les Waveforms,
  - ➡ Il est toutefois de posséder une simulation de référence...
  - ➡ Insuffisant en vérification
    - Comment savoir si les différences observées sont vraiment significatives ?
    - Par exemple, un léger retard d'une transition est elle problématique ?

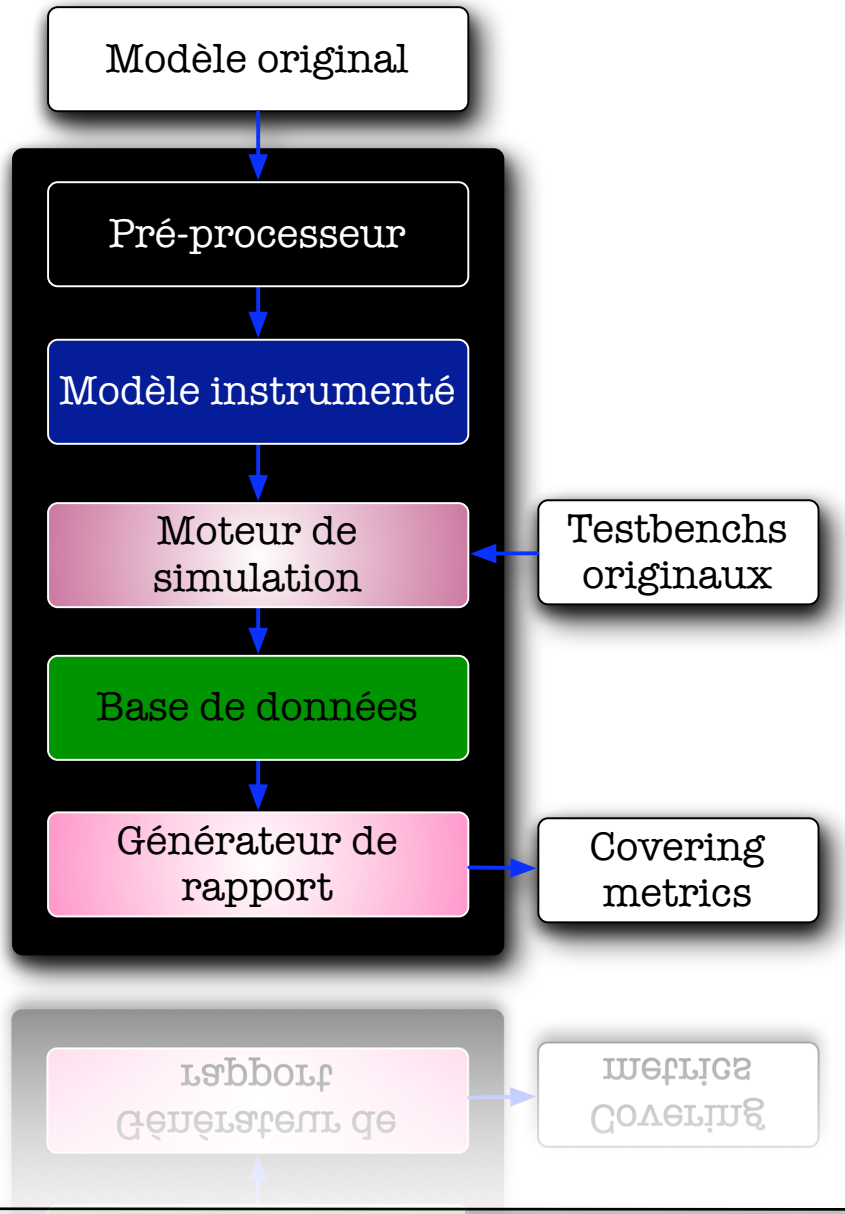


Visualisateur de signaux

Les outils d'aide à la vérification (les outils de couverture de code)

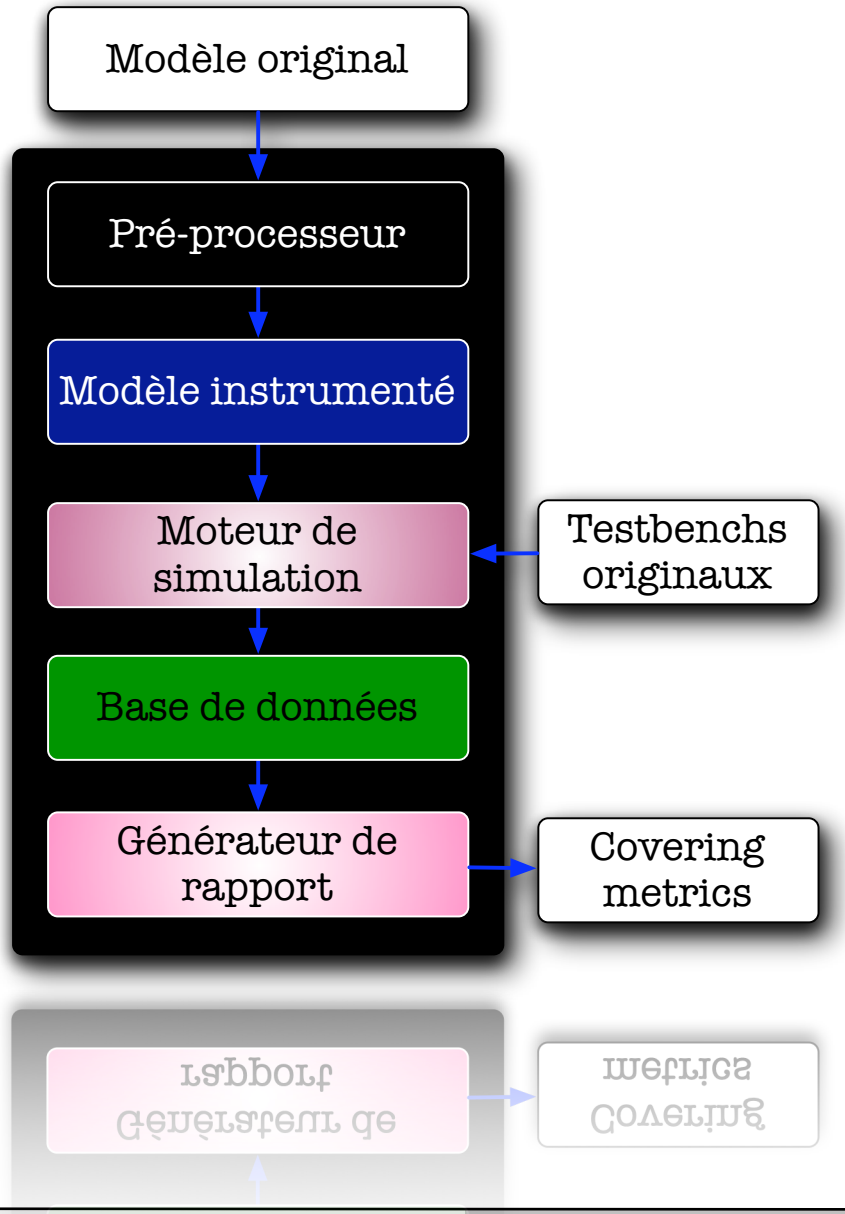
La couverture de code (Code coverage)

- Peut-on garantir qu'un design est correct s'il passe avec succès tous les testbenches ?
  - ➡ Un élément de réponse :
    A-t-on exécuté toutes les parties du code avec ces testbenches ?
- Les outils de Code Coverage permettent de savoir quelles parties du code ont été « exécutées » et surtout lesquelles ont été « non exécutées »
  - ➡ Instrumentation du code source



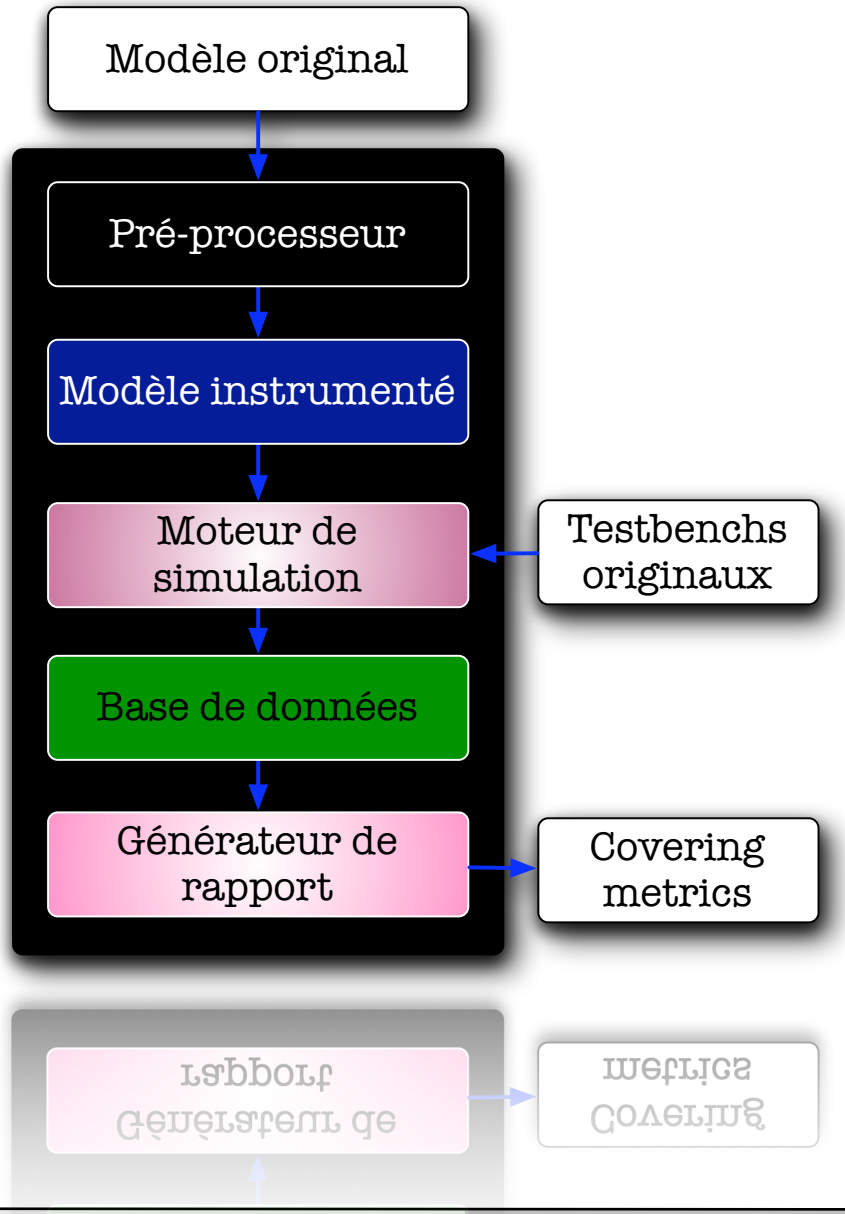
La couverture de code (Code coverage)

- Le code permettant l'instrumentation est ajouté automatiquement (typiquement des write pour vérifier le passage dans les différentes constructions du code)
- Différents rapports sont collectés/analysés et permettent de déterminer si l'ensemble des testbenches couvre ou non le code en cours de vérification



La couverture de code (Code coverage)

- Différentes métriques possibles :
  - Niveau lignes ou blocs (Statement Coverage)
  - Niveau chemins (Path Coverage)
  - Niveau expressions (Expression Coverage)
- L'utilisation des techniques de couverture implique :
  - L'achat ou l'utilisation d'outils spécifiques (à ajouter au flot de conception),
  - Des pénalités temporelles (ralentissement des simulations).



Statement coverage (Code coverage)

- Lors de la compilation du modèle des instructions sont rajoutées au code simulé ou exécuté,
- L'exécution de chaque ligne ou de chaque bloc (voir plus loin) du code est mémorisée,
- En plus de la mémorisation, il est possible d'obtenir le décompte des exécutions,

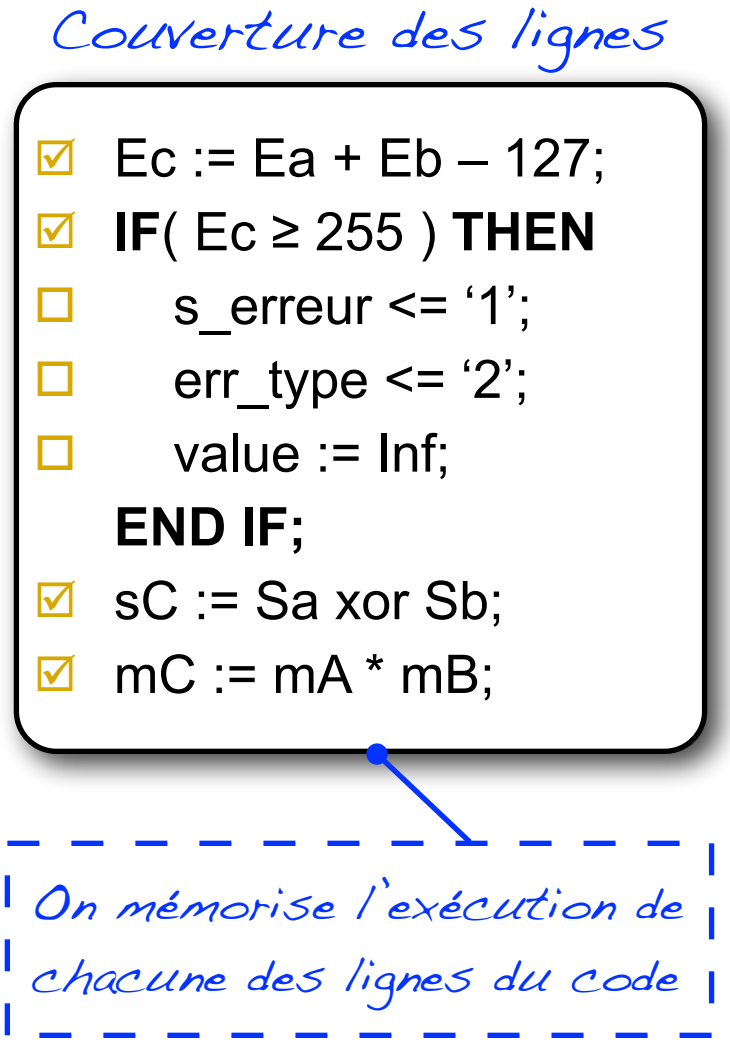
```
Ec := Ea + Eb – 127;
IF( Ec ≥ 255 ) THEN
    s_erreur <= '1';
    err_type <= '2';
    value := Inf;
END IF;
sC := Sa xor Sb;
mC := mA * mB;
```

Statement coverage (Code coverage)

- Analyse du résultat obtenu
  - Une partie du code n'est pas exécutée
  - Cette condition peut-elle vraiment se produire * (code mort) ou bien l'a-t-on oubliée dans le testbench ?
  - Comment faire pour exécuter cette partie ?
- L'objectif est d'avoir un taux de couverture du code de 100% **

* Le test d'une condition peut au contraire être volontaire afin de vérifier qu'on ne va jamais dans certaines configurations (voir partie assertion)

** 100% pas toujours atteignable (ex. : remarque ci-dessus, ou bien : when others <= null mis dans un case par le concepteur à cause de l'exhaustivité obligatoire du case mais pas nécessairement utile fonctionnellement parlant)



Statement coverage (Code coverage)

- Analyse de l'exécution du code sous la forme de blocs de lignes dépendantes :
  - Association temporelle (ensemble de lignes délimitées par un(des) wait)
  - Association conditionnelle (ensemble de lignes délimitées par THEN et ELSE dans un IF, et WHEN dans un CASE ...)
  - Autres (fonctions, procédures)
- Gain de temps lors la simulation
  - Moins de code d'instrumentation à insérer et à exécuter,
  - Moins d'information à mémoriser.

Couverture des blocs

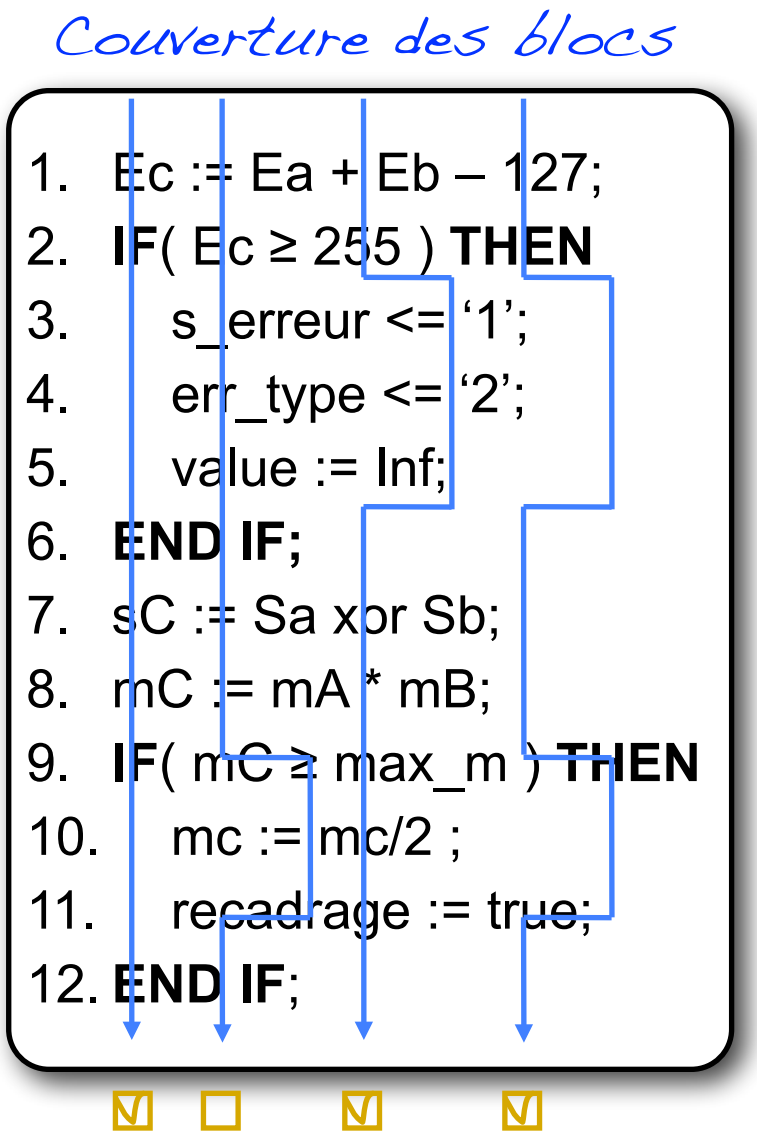
```
☑ Ec := Ea + Eb – 127;
☑ IF( Ec ≥ 255 ) THEN
☐     s_erreur <= '1';
      err_type <= '2';
      value := Inf;
  END IF;
☑ sC := Sa xor Sb;
☑ mC := mA * mB;
```

On considère un unique marqueur pour toutes les opérations appartenant à un bloc

Path coverage (Code coverage)

- L'objectif de cette analyse est de déterminer quels sont les chemins exécutés *
  - ➡ La complexité de l'analyse augmente de manière exponentielle avec le nombre de structures conditionnelles ! (utilisé pour les unités de conception de petite taille (≅ 100 lignes))
- Avec le Path Coverage, il est très difficile d'avoir un taux de couverture de 100%

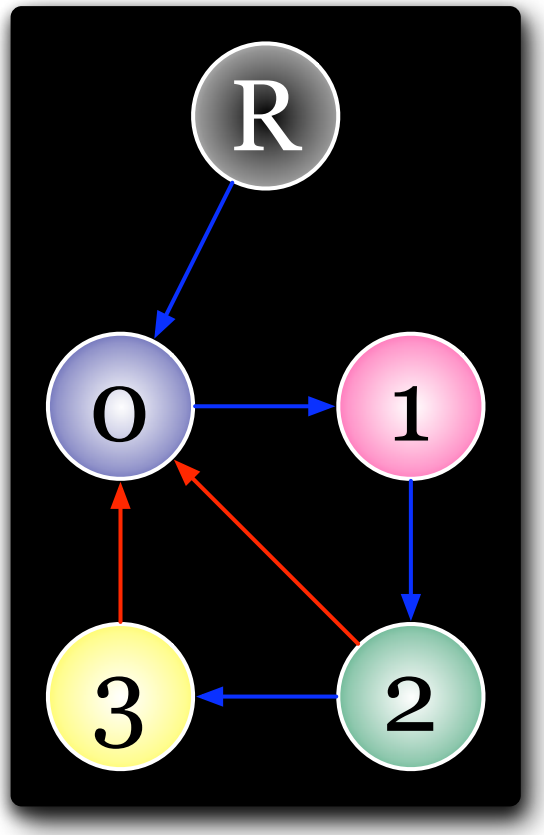
* En pratique, Path Coverage est souvent réalisée de pair avec Statement Coverage.



FSM coverage (Code coverage)

- Déterminer quelles sont les états visités (State Coverage) et les transitions franchies (Transition Coverage)
  - ➡ Facile de couvrir tous les états,
  - ➡ Difficile de franchir toutes les transitions !
- Exemples
  - ➡ Dans l'exemple à coté, tous les états de la FSM ont été atteint une fois,
  - ➡ la série (Reset => 1 => 2 => 3) a été réalisée,
  - ➡ Les transitions (2 => 0) et (3 => 0) n'ont pas été mise en oeuvre.

Couverture des états et des transitions



Validité du "Coverage"

- Les méthodes de « Code Coverage » permettent de savoir ce que le code de test que l'on a écrit exécuté dans le code à tester
  - ➡ Mais elles ne fournissent aucune garantie quant à la fiabilité et à la qualité de l'ensemble des testbenches utilisés,
    - Tous les lignes de code ont pu être executées avec des valeurs de données incohérentes ou hors plage de fonctionnement...
  - ➡ Fournissent une indication sur ce qui est vérifié (exécuté) grâce à l'ensemble des testbenches appliqués sur le design en cours de conception,
- Gestion des cas non couverts (si taux de couverture < 100%)
  - ➡ Oubli du concepteur lors de l'écriture du testbench ?
  - ➡ Normal car ils n'appartiennent pas aux spécifications du système ?
  - ➡ Quelle(s) condition(s) appliquer pour exécuter ce(s) cas ?

Les techniques de code coverage appliqué aux codes logiciels

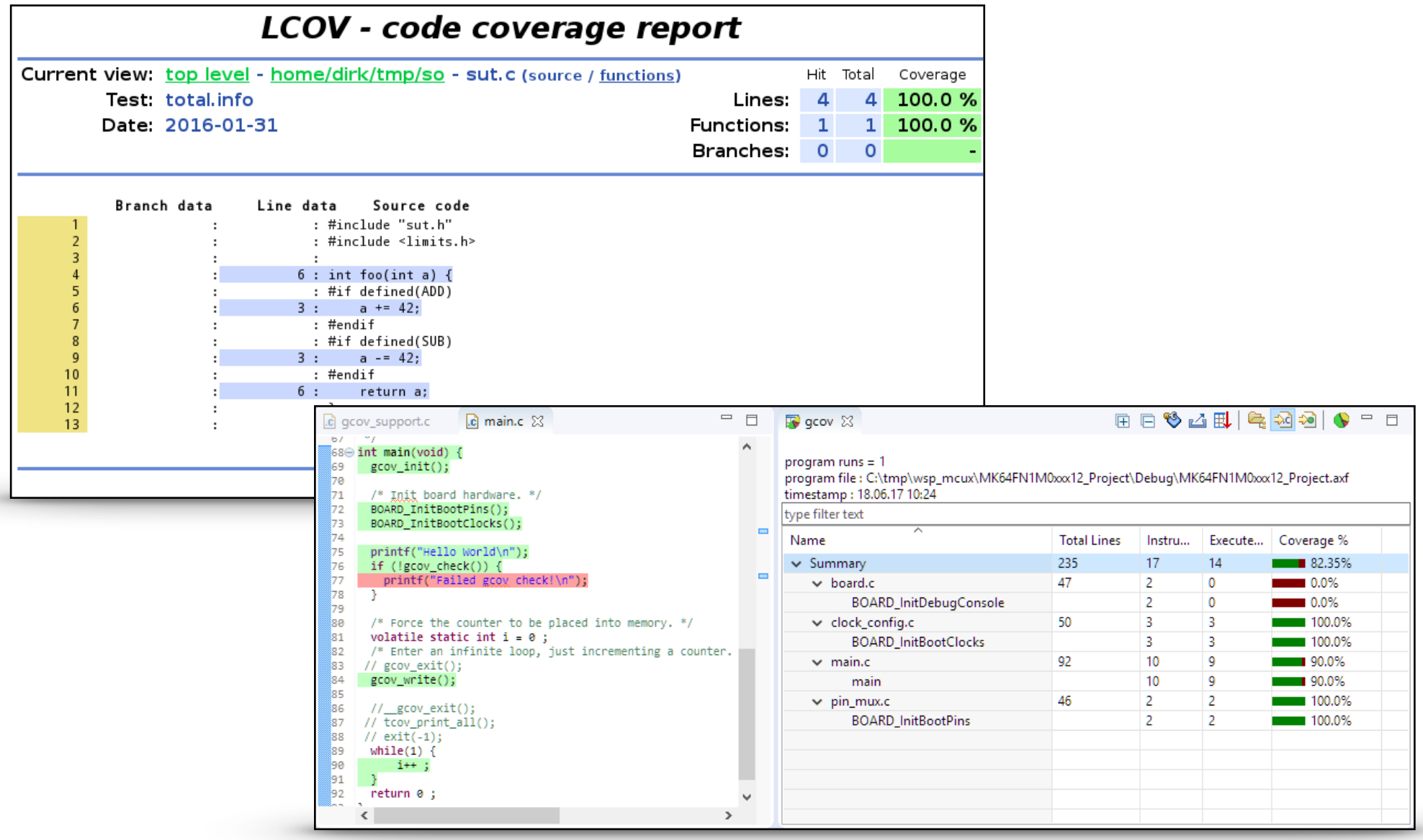
```c
#include <stdio.h>

int
main (void)
{
  int i;

  for (i = 1; i < 10; i++)
    {
      if (i % 3 == 0)
        printf ("%d is divisible by 3\n", i);
      if (i % 11 == 0)
        printf ("%d is divisible by 11\n", i);
    }

  return 0;
}
```

```
        #include <stdio.h>

        int
        main (void)
        {
     1    int i;

    10    for (i = 1; i < 10; i++)
            {
     9        if (i % 3 == 0)
     3          printf ("%d is divisible by 3\n", i);
     9        if (i % 11 == 0)
######          printf ("%d is divisible by 11\n", i);
     9      }

     1    return 0;
     1  }
```

```
gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

```
gcov cov.c
 88.89% of 9 source lines executed in file cov.c
Creating cov.c.gcov
```

Des outils plus évolués permettent d'accéder aux rapports



LCOV - code coverage report

Current view: top level - home/dirk/tmp/so - sut.c (source / functions)

Test: total.info

Date: 2016-01-31

	Hit	Total	Coverage
Lines:	4	4	100.0 %
Functions:	1	1	100.0 %
Branches:	0	0	-

```
      Branch data     Line data    Source code
 1                :            : #include "sut.h"
 2                :            : #include <limits.h>
 3                :            :
 4                :          6 : int foo(int a) {
 5                :            : #if defined(ADD)
 6                :          3 :     a += 42;
 7                :            : #endif
 8                :            : #if defined(SUB)
 9                :          3 :     a -= 42;
10                :            : #endif
11                :          6 :     return a;
12                :            :
13                :            :
```

gcov_support.c | main.c

```
68 int main(void) {
69   gcov_init();
70
71   /* Init board hardware. */
72   BOARD_InitBootPins();
73   BOARD_InitBootClocks();
74
75   printf("Hello World\n");
76   if (!gcov_check()) {
77     printf("Failed gcov check!\n");
78   }
79
80   /* Force the counter to be placed into memory. */
81   volatile static int i = 0 ;
82   /* Enter an infinite loop, just incrementing a counter.
83  // gcov_exit();
84   gcov_write();
85
86   //__gcov_exit();
87  // tcov_print_all();
88  // exit(-1);
89   while(1) {
90       i++ ;
91   }
92   return 0 ;
```

gcov

program runs = 1
program file : C:\tmp\wsp_mcux\MK64FN1M0xxx12_Project\Debug\MK64FN1M0xxx12_Project.axf
timestamp : 18.06.17 10:24

type filter text

Name	Total Lines	Instru...	Execute...	Coverage %
Summary	235	17	14	82.35%
board.c	47	2	0	0.0%
BOARD_InitDebugConsole		2	0	0.0%
clock_config.c	50	3	3	100.0%
BOARD_InitBootClocks		3	3	100.0%
main.c	92	10	9	90.0%
main		10	9	90.0%
pin_mux.c	46	2	2	100.0%
BOARD_InitBootPins		2	2	100.0%

Les outils d'aide à la vérification (les outils de simulation)

Les simulateurs, les outils les plus familiers...

- Se nomment simulateurs car sont bornés à approximer la réalité (certains paramètres physiques sont simplifiés ou ignorés)
  - ➡ Ex.: avec un simulateur logique, un signal peut valoir {0, 1, Z, U}; en réalité, des valeurs analogiques (tension) entre [0, "5"V].
- Dans ce monde « restreint », le simulateur exécute une description. Cette description est définie à l'aide d'un langage borné avec des sémantiques précises.
  - ➡ Si la description ne reflète pas précisément la réalité, comment peut on savoir que ce que l'on simule est différent de ce qui sera finalement réalisé (fabriqué) ?
- Cette approche nécessite la définition de stimulus.
  - ➡ Au concepteur de fournir les "bons" stimulus (ceux qui permettront de dire si oui ou non le design est correct)
  - ➡ Le simulateur ne connaît pas les intentions du concepteur. La notion de validation (design correct ou non) n'est connue que du concepteur qui doit interpréter les résultats post-simulation.

Les problèmes liés à l'utilisation des simulateurs

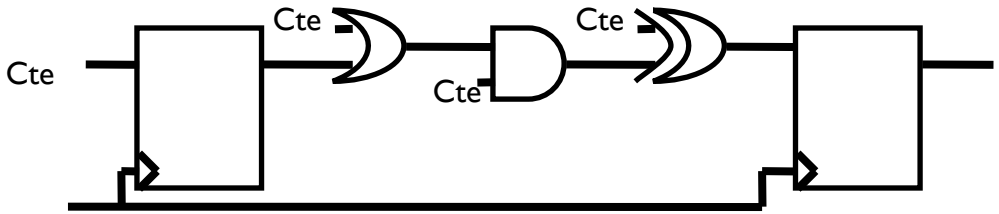
- Précision et fiabilité des modèles physiques utilisés dans les simulateurs :
  - Ce qui n'est pas considéré dans le modèle ne peut pas être prédit !
  - La précision des résultats est généralement fonction de la complexité du modèle (ex. modèles météo…),
  - Les phénomènes de durée inférieure au pas de simulation (simulateurs discrets) ne peuvent pas être perçus et donc considérés...
- La vitesse de simulation est généralement très faible par rapport au circuit réel (dépend de la complexité du modèle de simulation),
  - Le courant électrique traverse un circuit contenant quelques millions de portes en 1 seconde,
  - Le simulateur est implémenté sur un processeur pouvant exécuter un million d'instructions par seconde,
  - Le temps de simulation de certains circuits de niveau RTL peut facilement atteindre une journée, une semaine, des mois …

Les simulateurs Event-driven

- Simulateur « de base » : on exécute à chaque cycle l'ensemble des composants du circuit, qu'ils le nécessitent ou pas.
  - ➡ Pas très efficace (rapidité) + problème de la durée d'un cycle
- Les simulateurs de type Event'driven sont des simulateurs basés sur la propagation d'événements entre « les composants »
  - ➡ On exécute uniquement les composants qui peuvent changer d'état (inutile d'exécuter une porte logique si ses entrées n'ont pas changé)
  - ➡ Utilisation de sémantiques supplémentaires dans le langage de description
- Exemple
  - ➡ Exécution SI clock'event OR input'event

Les simulateurs Cycle-based

- Dans le cas de circuits synchrones, il n'est pas nécessairement utile d'oserver les évènements intermédiaires,
  - ➡ Par exemple, plusieurs portes logiques combinatoires en série entre la sortie et l'entrée de 2 flip-flops synchronisés par la même horloge + seules les sorties des 2 registres sont exploitées. Inutile de faire une simulation event-driven).
- Gain de temps de simulation possible



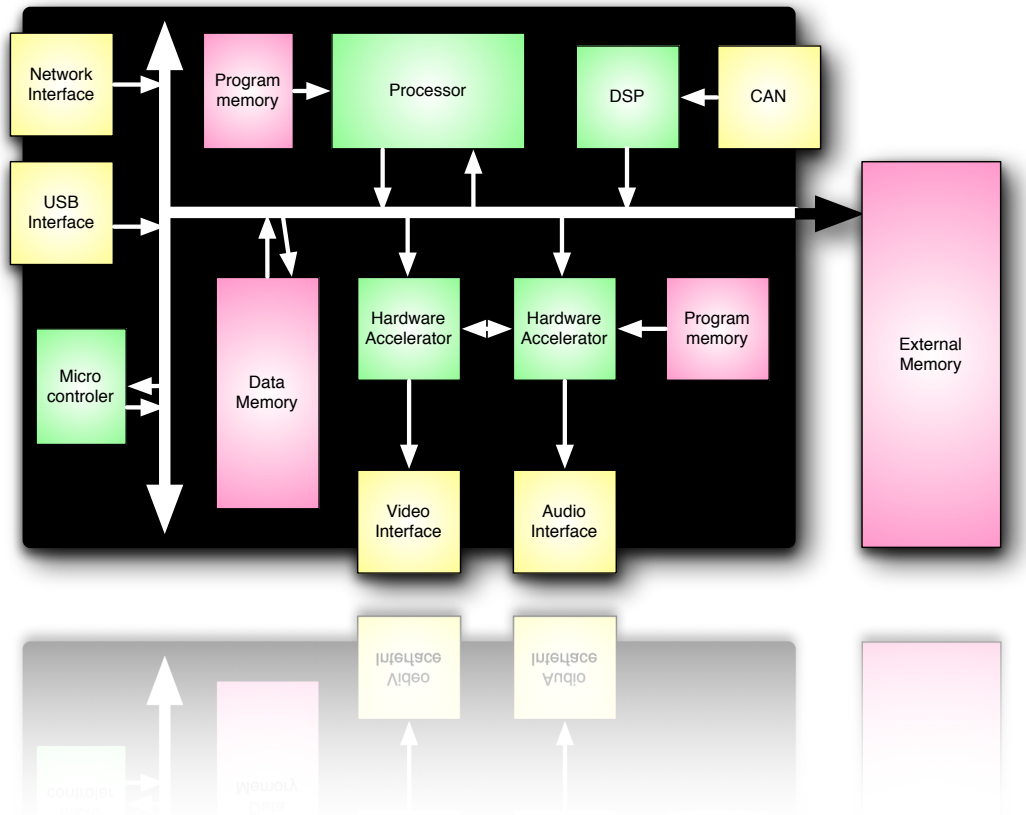
- 2 étapes pour la simulation cycle-based :
  - ➡ Calcul logique (simplification) des fonctions combinatoires concernées
  - ➡ Mise à jour des sorties des flip-flops au front actif de l'horloge à partir de la valeur retournée par l'étape de calcul logique

Les simulateurs Cycle-based

- Avantage
  - Plus rapides que event-based si l'étape de calcul logique prend moins de temps que la simulation des événements intermédiaires (très souvent vrai) et qu'il y a beaucoup de registres à changer de valeur au front actif
- Inconvénients
  - Perte complète des informations temporelle (temps, délais) (entre autre, setup et hold times assumés comme respectés implicitement)
  - Réservés aux circuits synchrones (le seul événement important est le front actif de l'horloge).
- En pratique
  - un circuit contient des parties synchrones et des parties asynchrones. Les simulateurs cycle-based sont généralement intégrés dans un simulateur event-driven. Les partie synchrones sont simulées en utilisant un algorithme cycle-based, les autres avec le simulateur event-driven, ce qui permet de (co)simuler l'ensemble efficacement.

Les simulateurs – Co-simulation

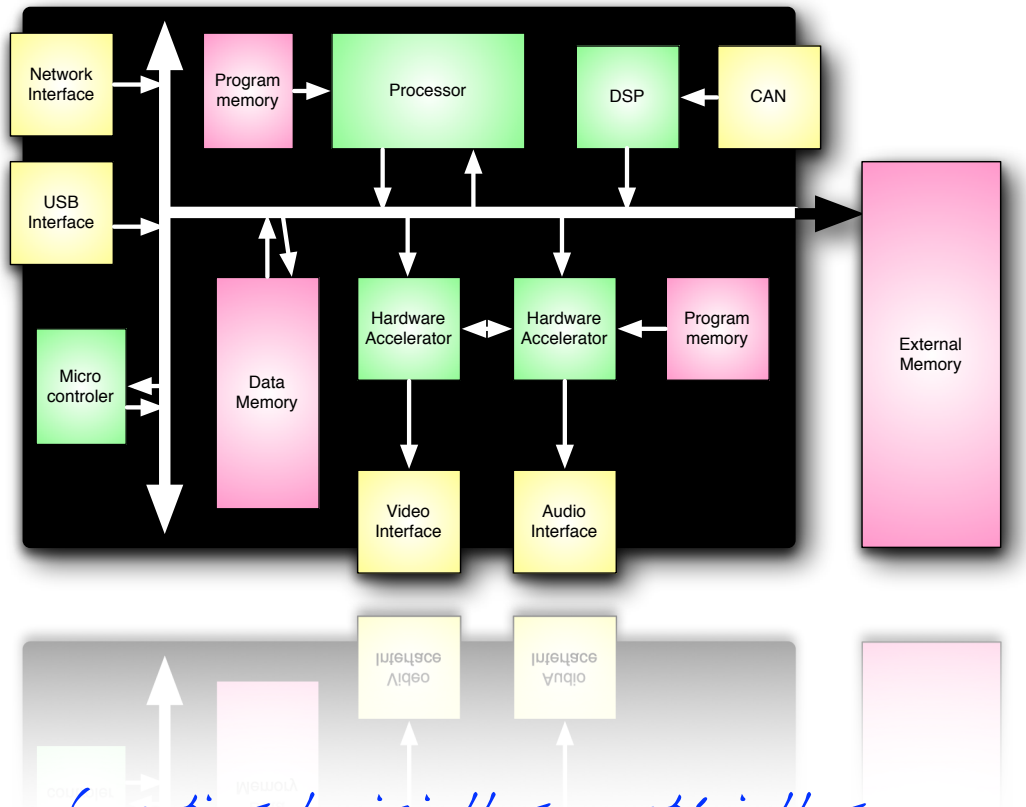
- Les circuits actuels (type SoC) contiennent différentes parties :
  - Hardware & Software (+ OS)
  - Numérique & Analogique
  - Différentes technologies
- Ces circuits ne peuvent être simulés à l'aide d'un seul simulateur (≠ langages, …)
- Chaque simulateur va gérer ses blocs élémentaires lors de la simulation du système



Un système actuel est un savant mélange entre des parties logicielles, matérielles, numériques et analogiques.

Les simulateurs – Co-simulation

- Cohabitation d'au moins 2 simulateurs de types différents*
- Les simulateurs doivent progresser au même rythme
  - ➡ Tous se calent sur le plus lent,
- Surcoût en terme de communication/synchronisation entre les simulateurs (E/S) :
  - ➡ Besoin de traduire/convertir les événements et les valeurs d'un simulateur à l'autre,
    - Ambiguïtés possibles (ex.: comment traduire une tension (analog.) en une valeur (logique) ?

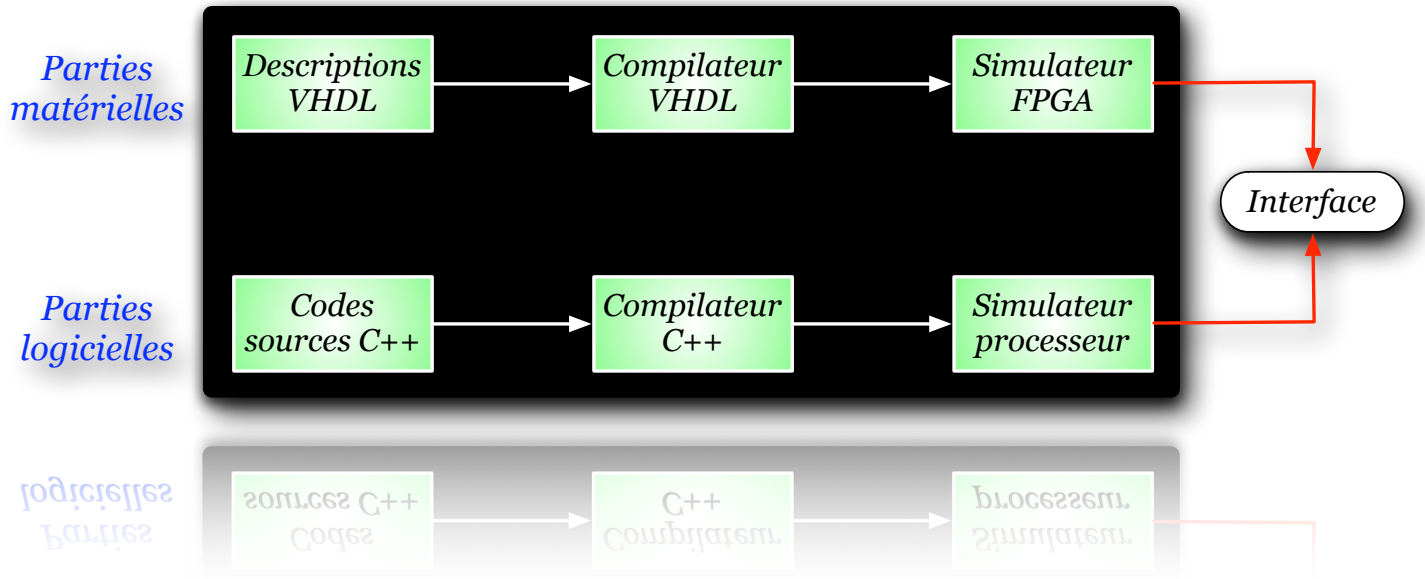


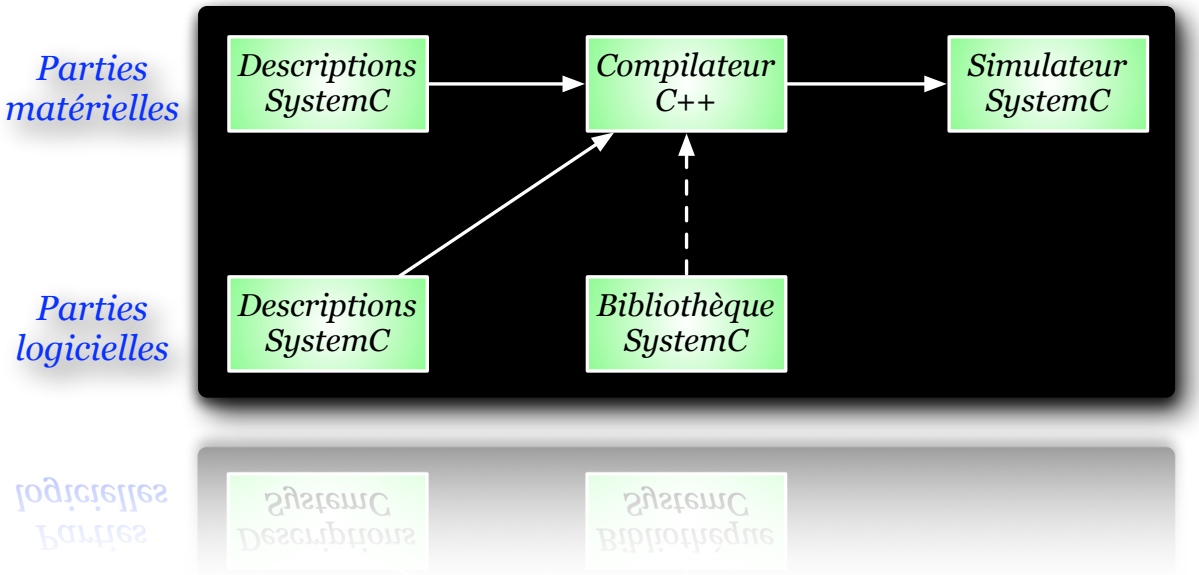
(parties logicielles, matérielles, numériques et analogiques)

* ≠ simulateurs multi-langages (compilation des différents langages en une représentation interne unique, un seul simulateur)

Simulateurs et conception conjointe

Co-simulation de modèles décrits dans des langages différents



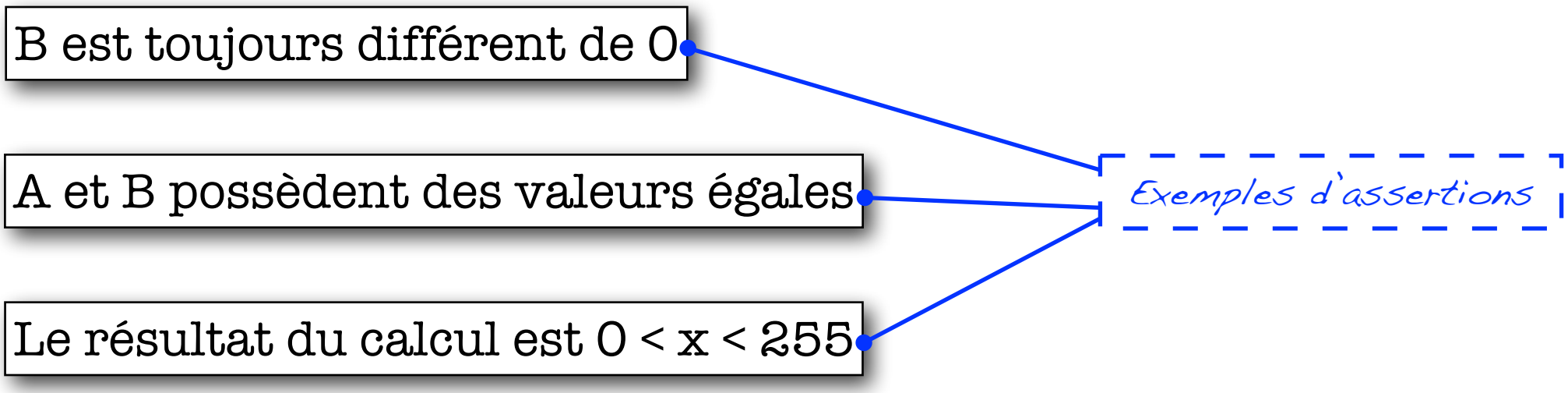


Co-simulation de multiples modèles décrits dans le langage SystemC

Les outils d'aide à la vérification (les assertions fonctionnelles)

Introduction aux assertions fonctionnelles

- Les mécanismes d'assertion permettent de vérifier que des hypothèses émises lors de la conception d'un système sont toujours respectées lors de son exécution,
  - ➡ Méthodes de vérification provenant du développement logiciel (assertions natives en C, C++, Java, …),
  - ➡ Traduction de assertion : affirmation ou hypothèse



L'intérêt des assertion dans les étapes de vérification

- Les assertions permettent de valider le comportement de l'application lors de l'exécution et de gérer les erreurs dynamiques,
  - ➡ Une erreur stoppe l'exécution du programme avec indication de la cause de l'erreur + localisation,
  - ➡ Les assertions sont utilisées uniquement dans le cadre du développement et de la mise au point : l'utilisateur n'a pas besoin d'observer les problèmes qui ne le concernent pas,
- Elles disparaissent à la compilation lorsque l'on utilise les optimisations du compilateur,
- Outil indispensable lors des phases de conception (inadapté autrement).
- Des mécanismes de gestions d'assertions sont disponibles dans la majorité des langages de conception matériel et logiciel (VHDL, SystemC, Java, C++, C, etc.).

Mise en oeuvre des assertions en C/C++

```c
#include <iostream>
#include <stdio.h>
#include <assert.h>

int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}

int main (int argc, char * const argv[]) {
    ma_fonction(2, 0);
    return 0;
}
```

Voici un exemple de code source dans lequel nous avons introduit une assertion afin de réaliser une hypothèse de fonctionnement.

Exemple d'utilisation des assertions en langage C++

Compilation normale du programme à l'aide de GCC :

gcc mon_prog.c -o mon_prog

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 0);

```
The Debugger has exited with status 0.
[Session started at 2007-12-12 15:12:05 +0100.]
Lancement de la division...
Assertion failed: (b != 0), function ma_fonction, file
      /Users/legal/XCode/Essais_code/main.cpp, line 32.

The Debugger has exited due to signal 6 (SIGABRT).
```

Exécution du binaire :

./mon_prog

Exemple d'utilisation des assertions en langage C++

Compilation normale du programme à l'aide de GCC :
gcc mon_prog.c -o mon_prog

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 2);

```
The Debugger has exited with status 0.
[Session started at 2007-12-12 15:12:04 +0100.]
Lancement de la division...
Fin de la division...
```

Exécution du binaire :
./mon_prog

Exemple d'utilisation des assertions en langage C++

Compilation avec désactivation des assertions à l'aide de GCC :

```
gcc mon_prog.c -o mon_prog -NDEBUG
```

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 0);

```
[Session started at 2007-12-12 18:01:16 +0100.]
Lancement de la division...
Fin de la division...

The Debugger has exited due to signal 8 (SIGFPE).
```

Exécution du binaire :

./mon_prog

=> Plantage !

Utilisation des assertions en langage Java

```
public void ma_fonction(int a, int b ){
   assert a >= 0 && b >= 0;
   int n = a;
   assert n + b == a + b;
   int som = b;
   assert n + som == a + b;
   while (n > 0) {
     assert n + som == a + b : "L'invariant n'est pas vérifié";
     n = n - 1;
     som = som + 1;
     assert n + som == a + b : "L'invariant n'est pas vérifié";
   }
   assert n <= 0 && n + som == a + b : "L'opération a échoué";
}
```

Par défaut à l'exécution les assertions sont désactivées, il faut passer un paramètre à l'interpréteur :

=> java -enableassertions -jar Mon_Application.jar

=> java -ea -jar Mon_Application.jar

Exemple d'utilisation des assertions en langage Java

```java
public void ma_fonction(int a, int b ){
   assert a >= 0 && b >= 0;
   int n = a;
   assert n + b == a + b;
   int som = b;
   assert n + som == a + b;
   while (n > 0) {
     assert n + som == a + b : "L'invariant n'est pas vérifié";
     n = n - 1;
     som = som + 1;
     assert n + som == a + b : "L'invariant n'est pas vérifié";
   }
   assert n <= 0 && n + som == a + b : "L'opération a échoué";
}
```

ma_fonction(2, -1);

```
> java -ea -jar Mon_Application.jar
    Exception in thread "main" java.lang.AssertionError
       at graphbox.Main.ma_fonction(Main.java:113)
       at graphbox.Main.main(Main.java:143)
```

Exemple d'utilisation des assertions en langage Java

```java
public void ma_fonction(int a, int b ){
   assert a >= 0 && b >= 0;
   int n = a;
   assert n + b == a + b;
   int som = b;
   assert n + som == a + b;
   while (n > 0) {
     assert n + som == a + b : "L'invariant n'est pas vérifié";
     n = n - 1;
     som = som + 1;
     assert n + som == a + b : "L'invariant n'est pas vérifié";
   }
   assert n <= 0 && n + som == a + b : "L'opération a échoué";
}
```

ma_fonction(x, y);

```
> java -ea -jar Mon_Application.jar
  Exception in thread "main" java.lang.AssertionError: L'invariant n'est pas vérifié
       at graphbox.Main.ma_fonction(Main.java:122)
       at graphbox.Main.main(Main.java:143)
```

Les assertions « statiques » en C++

Static Assertion

Performs compile-time assertion checking

Syntax

static_assert (*bool_constexpr* , *message*)	(since C++11)
static_assert (*bool_constexpr*)	(since C++17)

Les assertions statiques sont vérifiées lors de la compilation du programme

```cpp
#include <iostream>

using namespace std;

#define N 3

int main()
{
    // Verification de la machine hote
    static_assert( sizeof(char) == 1 );

    // Verification des parametres de configuration
    static_assert( N%2 == 0 );

    return 0;
}
```

Mise en oeuvre des assertions en VHDL

- L'instruction **ASSERT** en VHDL est une instruction de déboggage.
  - Elle permet de vérifier si une condition est vraie, dans le cas contraire, reporte un message sur l'écran du simulateur.
  - **ASSERT** est très utile afin de vérifier les violations de timing.
- Listes des différents niveaux de "gravité"
  - **NOTE** : utilisé pour information seulement
    - "Note : Chargement de données d'un fichier"
  - **WARNING** : utilisé pour fournir une information sur une erreur en instance
    - "Warning : Détection d'un pic"
  - **ERROR** : utilisé pour information seulement
    - "Error :Violation du temps d'initialisation"
  - **FAILURE** : reporte une grosse erreur
    - "Failure : Ligne RESET instable"

Exemple de mise en oeuvre des assertions en VHDL

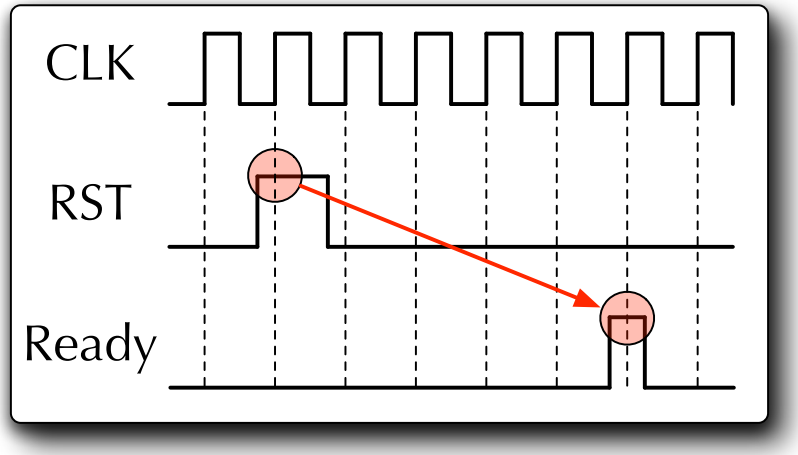
```
ASSERT condition
    [REPORT string] [SEVERITY severity_level];
```

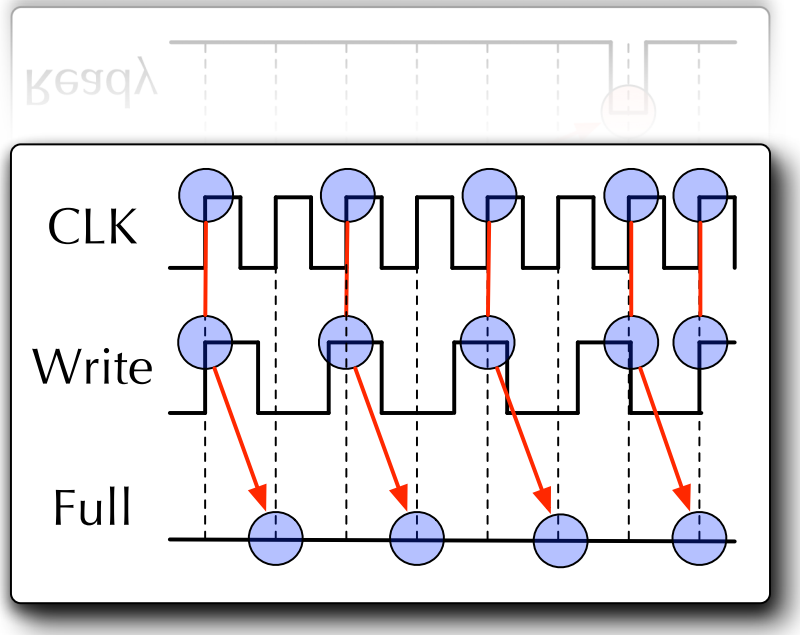
Les assertions sont générées dans ModelSim et ignorées dans ISE et Quartus.

```
check_setup: PROCESS (clk, d)
BEGIN

   IF (clk'EVENT AND clk='1') THEN
      ASSERT ( UNSIGNED(A) < 10 OR UNSIGNED(B) > 20 )
         REPORT "Erreur, dépassement des valeurs normalisées en entrée"
         SEVERITY ERROR;
   END IF;

END PROCESS check_setup;
```

Introduction au assertions dites matérielles

- Les assertions matérielles ont été développées afin de vérifier les hypothèse de conception,
- Quelques exemples
  - Un cycle après un reset, le composant, repositionne bien son signal ready à l'état haut 5 cycles après l'envoi de l'impulsion
  - Vérification du non débordement d'une FIFO (notion temporelle pas nécessaire),
  - Vérification que 2 cycles après avoir reçu un signal REQ, le composant répond par un ACQ…





Exemple de mise en oeuvre des assertions en VHDL

```
ASSERT condition
    [REPORT string] [SEVERITY severity_level];
```

Les assertions sont générées dans ModelSim et ignorées dans ISE et Quartus.

```
check_setup: PROCESS (clk, d)
BEGIN

   IF (clk'EVENT AND clk='1') THEN      -- test si front montant de clk
      ASSERT d'STABLE(setup_time)       -- regarde si d est stable pendant setup_time
         REPORT "Setup Violation..."    -- affiche un message d'avertissement si pas de stabilité
         SEVERITY WARNING;
   END IF;

END PROCESS check_setup;
```

Exemple de langage dédié aux assertions (PSL)

Le **Property Specification Language** (PSL) (en français : *Langage de spécification par propriétés*) est basé sur le langage Sugar d'IBM. Il a été approuvé par l'organisme Accellera en mai 2003, et par l'IEEE en septembre 2004.

Cette unité de verification (vunit) permet de vérifier sur front montant de CLK qu'on n'a jamais SCLK=0 quand CS_N=1:

```
vunit checker_spi(top)
 default clock : posedge(CLK);
 property p0 : never(!SCLK && CS_N);
 d0 : assert p0;
```

Cette unité de verification (vunit) permet de vérifier sur front montant de CLK qu'on a 8 coup d'horloge SCLK après le pasage à 0 de CS_N:

```
vunit checker_spi(top)
 default clock : posedge(CLK);
 sequence fe_CS_N : {CS_N;!CS_N};
 property p0 : always({fe_CS_N} |→ {SCLK;{!SCLK;SCLK}[*8]});
 d0 : assert p0;
```

http://fr.wikipedia.org/wiki/Property_Specification_Language

Utilité des assertions vis-a-vis des autres tests

- Les assertions dans les flots de compilation,
  - L'avantage des assertions dans les flots logiciels provient de leur gestion au niveau de la compilation,
    - Lors de la compilation, il est possible à l'aide d'une option du compilateur de les activer ou tout bonnement de les supprimer !
  - Lorsque les étapes de conception, test, validation sont terminées, il suffit d'activer l'option pour obtenir un binaire "propre" et performant,
- Les assertion dans les flots de synthèse,
  - La gestion des assertions se fait via les simulateurs matériels utilisés (voir à l'aide d'outils tiers),
  - Les outils de synthèse logique (ISE, Quartus) ignorent ces instructions lors de la synthèse évitant la modification manuelle des codes sources,
    - Suppression des risques d'introduction d'erreurs de codage...

Le cas particulier des assertions d'implémentation

- Elles sont spécifiées par le concepteur (designer)
  - Permettent d'encoder formellement les hypothèses faites durant la phase de conception des composants (le concepteur précise par exemple que le diviseur doit toujours être différent de la valeur 0)
  - Elles ne permettent pas de détecter des divergences entre la spécification et l'implémentation (ne sont pas faites pour ça)`
- Hypothèse de fonctionnement
  - La taille de la matrice dont on doit calculer le déterminant ne peut jamais être (0x0) ni (1x1),
  - La chaîne de caractères contenant le nom de l'utilisateur a une taille comprise entre 0 < 255 (utilisation d'une allocation statique),
  - La taille du fichier que l'application va charger en mémoire cache est toujours inférieure à 2 méga-octets.

Le cas particulier des assertions de spécification

- Elles sont spécifiées par l'équipe de vérification
  - Permettent d'encoder formellement les hypothèses et les contraintes exprimées dans les spécifications comme par exemple :
    - Latence maximum d'un circuit, cadence de production des sorties,
    - Ordre de validation des signaux lors d'un Hand-Shake, etc.
  - Elles visent à détecter les erreurs fonctionnelles présentes dans le circuit conçu,
  - Avec la complexité des circuits actuels, stratégie de type white-box (niveau bloc) => interaction forte nécessaire avec les designers,
- Hypothèse du cahier des charges
  - Le circuit doit produire une données tous les 1/24 de secondes afin de respecter la cadence du système,
  - Si le composant gérant l'ABS reçoit un front montant des freins, il doit au maximum 10 cycles après transmettre l'ordre adéquate aux commandes des disques.

Il existe d'autres manière d'exploiter les assertions

- Simulation assertions
  - Les assertions vont être vérifiées lors de l'exécution (simulation) du modèle
    NB : ASSERT transparent à la synthèse
- Formal assertions
  - Inconvénient des assertions de simulation : la simulation peut prouver la présence de bugs mais ne peut pas prouver leur absence (lié aux stimulus)
  - L'objectif des assertions formelles est de prouver mathématiquement que les hypothèses sont vérifiées (indépendamment de tesbenches)
  - Outils associés : Model Checkers ou Assertion Provers
    - Pour un compteur, l'outil fournira la suite d'état qui conduirait au non respect d'une hypothèse (au concepteur de savoir si cette séquence peut avoir lieu en pratique)
  - Utilisation en complément de la simulation : là où les conditions de simulation sont difficiles à créer pour vérifier un point particulier
  - Outils peu utilisés : équipes de conception peu familières avec ces techniques (principe radicalement différent de la simulation)

Les outils d'aide à la vérification (les outils d'analyse)

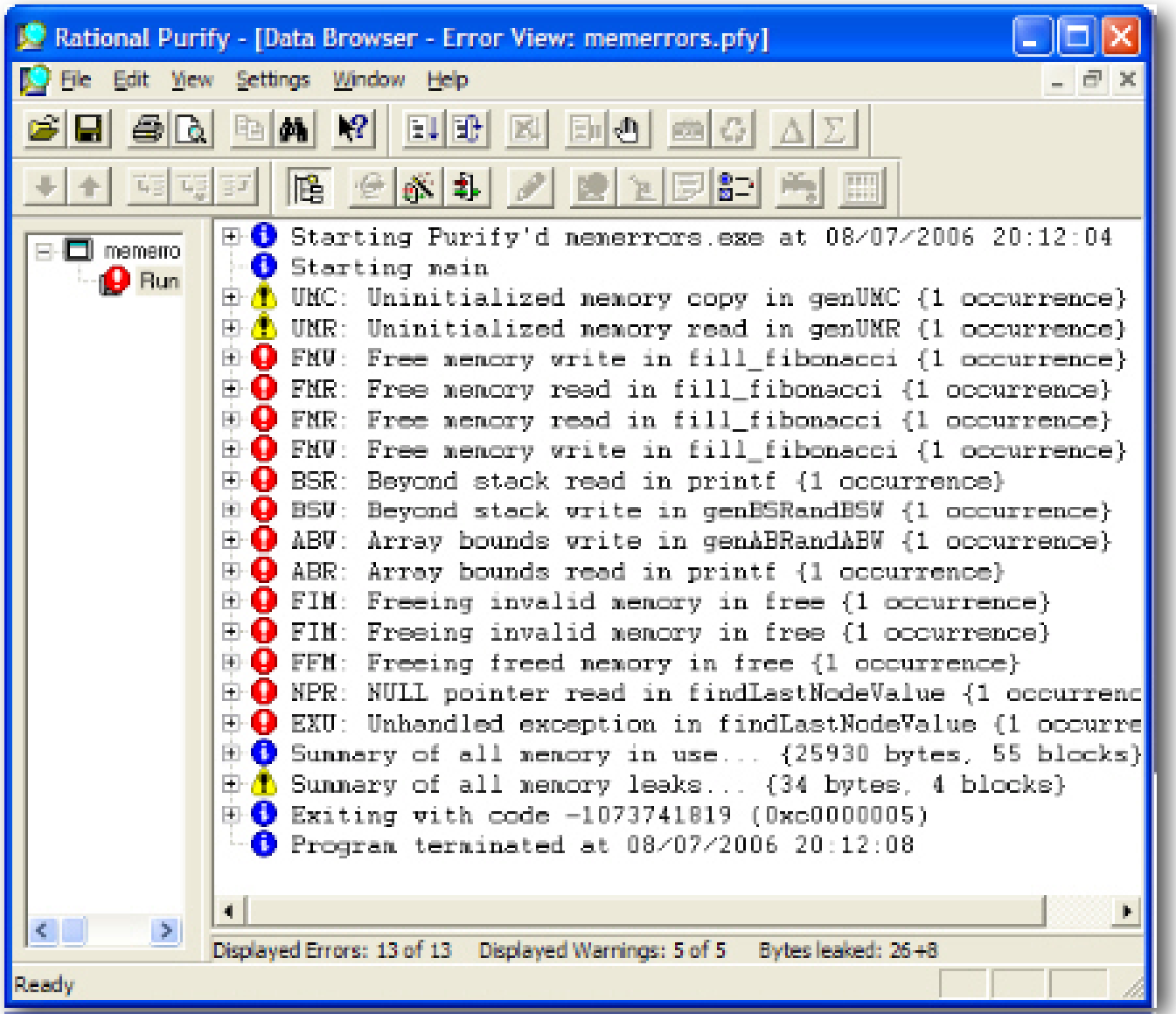
Les autres paramètres que l'on peut vouloir vérifier...

- Le vérification du fonctionnement d'un système ne se limite pas à ses entrées & sorties,
  - ➡ Fonctionnalité au sens algorithmique du terme.
- Il existe d'autres propriétés non-fonctionnelles que l'on souhaite pouvoir vérifier:
  - ➡ Temps d'exécution,
  - ➡ Consommation mémoire (et fuites),
  - ➡ Consommation d'énergie du système,
  - ➡ etc...
- Pour répondre à ces besoins, il existe un grand nombre d'outils qui diffèrent en fonction des langages utilisés et des systèmes cible.

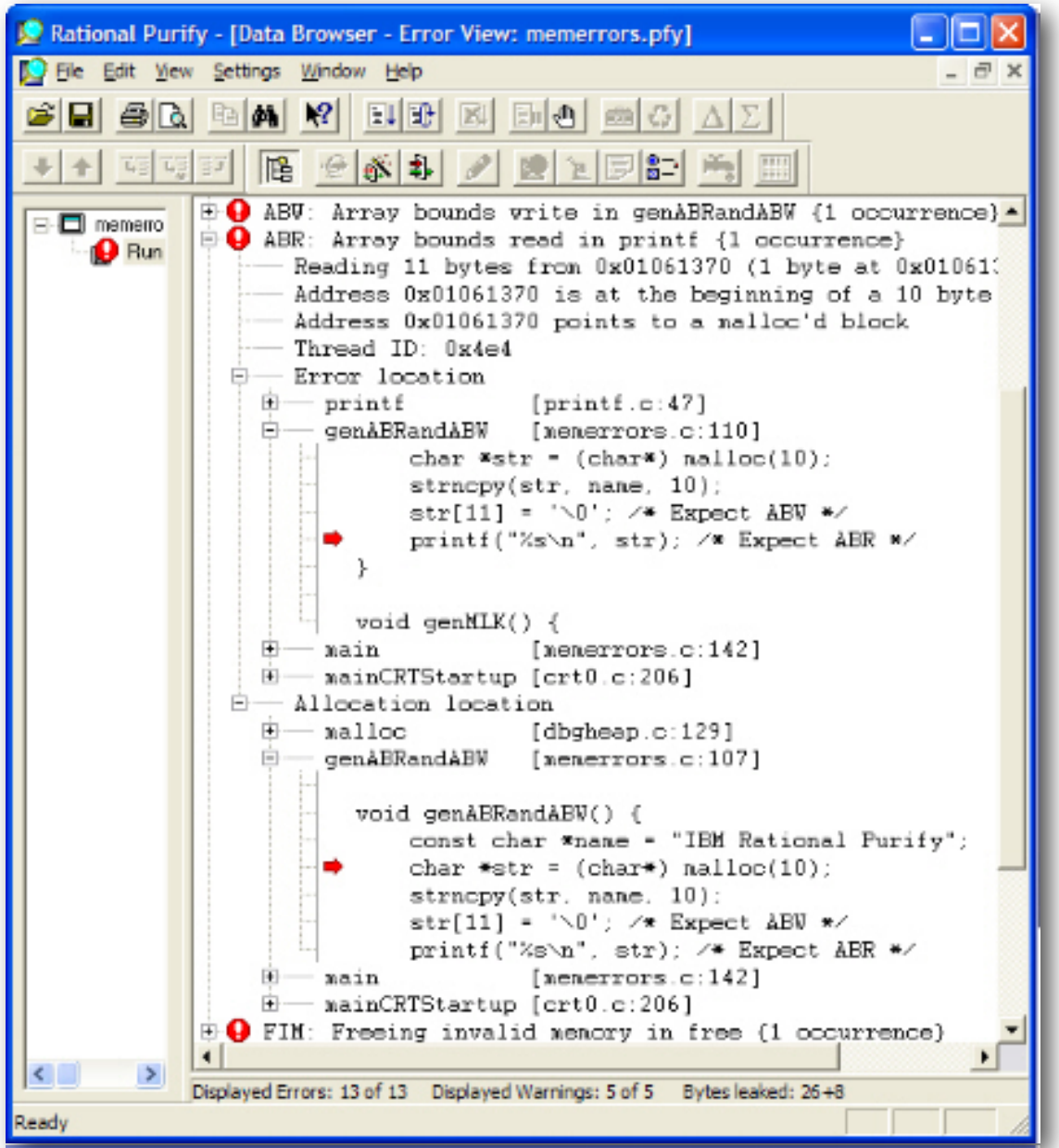
Etude de la consommation mémoire

- Les outils de couverture (Coverage) sont développés aussi dans le domaine de l'étude de la mémoire,
- Ce type d'outil permet :
  - Analyse et validation de la consommation mémoire maximum (contrainte) lors de l'exécution de l'application,
  - Analyse des fuites mémoires lors de l'execution (zone allouée et jamais libérée),
  - Analyse des débordements d'accès aux tableaux de données (utilisé afin de palier aux problèmes de sécurité : buffer over flow),
- Possibilité de détourner l'utilisation,
  - Validation du nombre d'objets crées en mémoire vis-a-vis du cahier des charges.
- Un outil gratuit et disponible sous Linux se nomme valgrind. Le leader du marché est Purify de Rationnal Software.

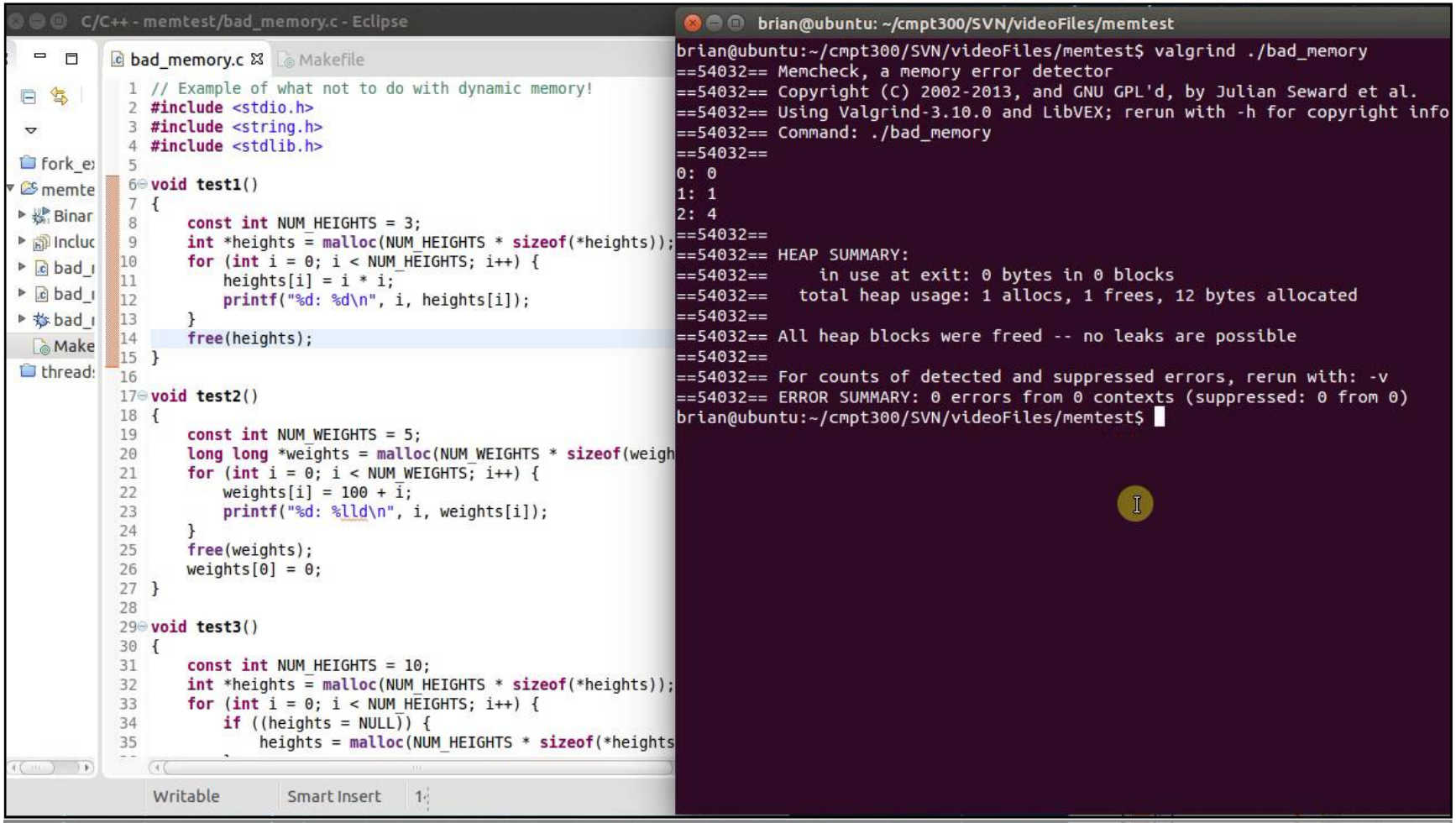
Exemple de rapport d'analyse (execution d'un programme)



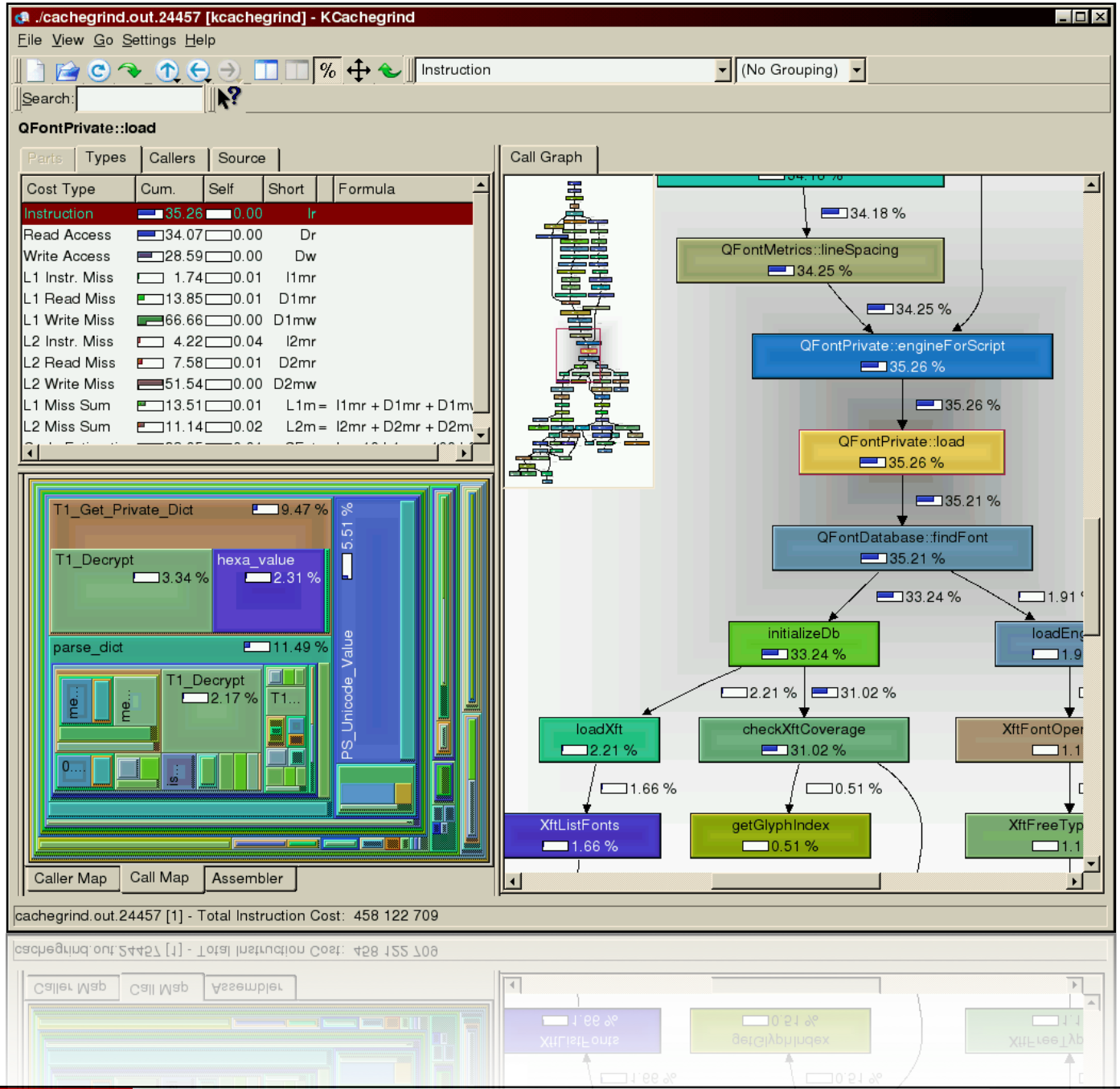
Exemple de rapport d'analyse (execution d'un programme)



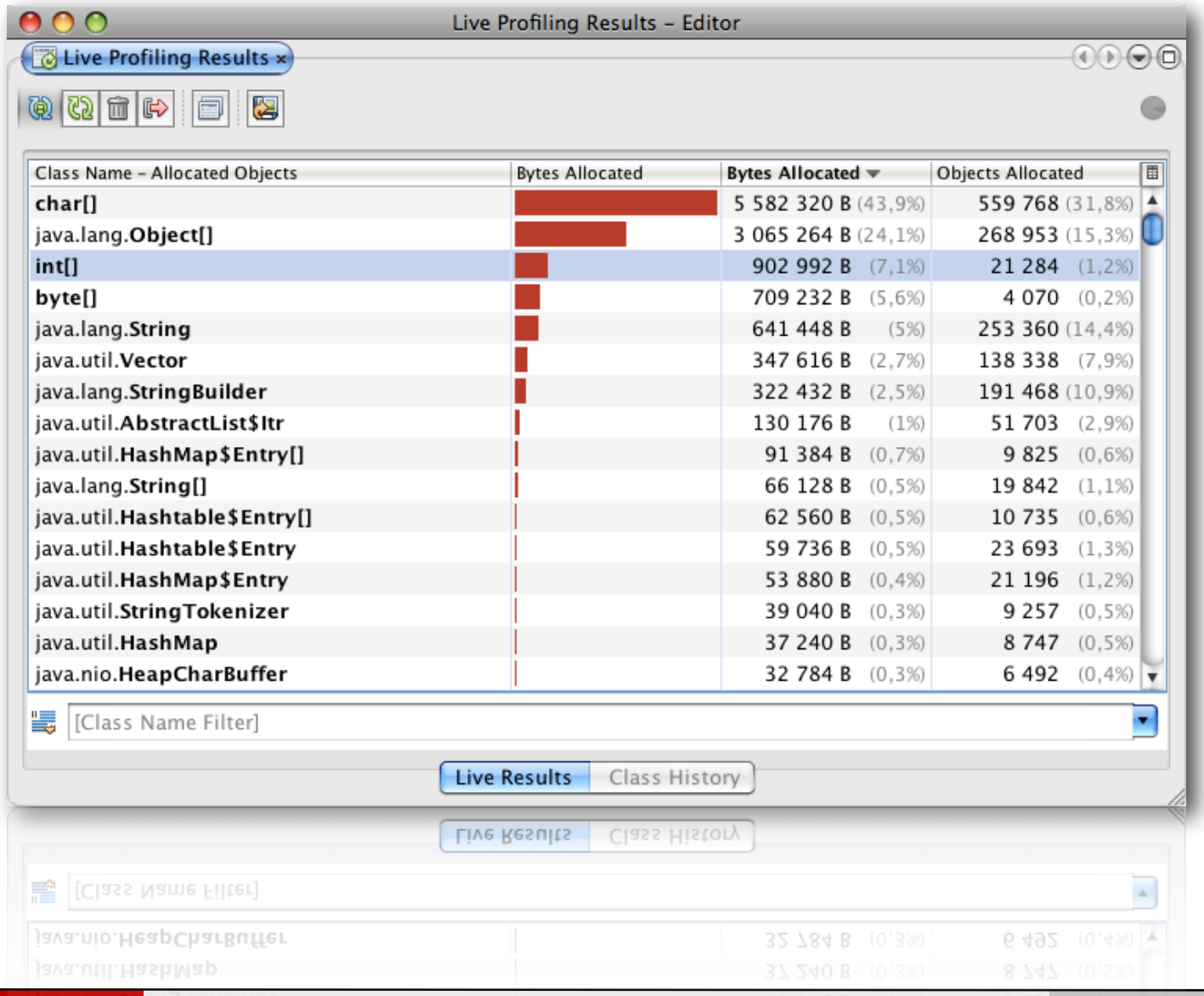
Valgrind - la référence Open-Source



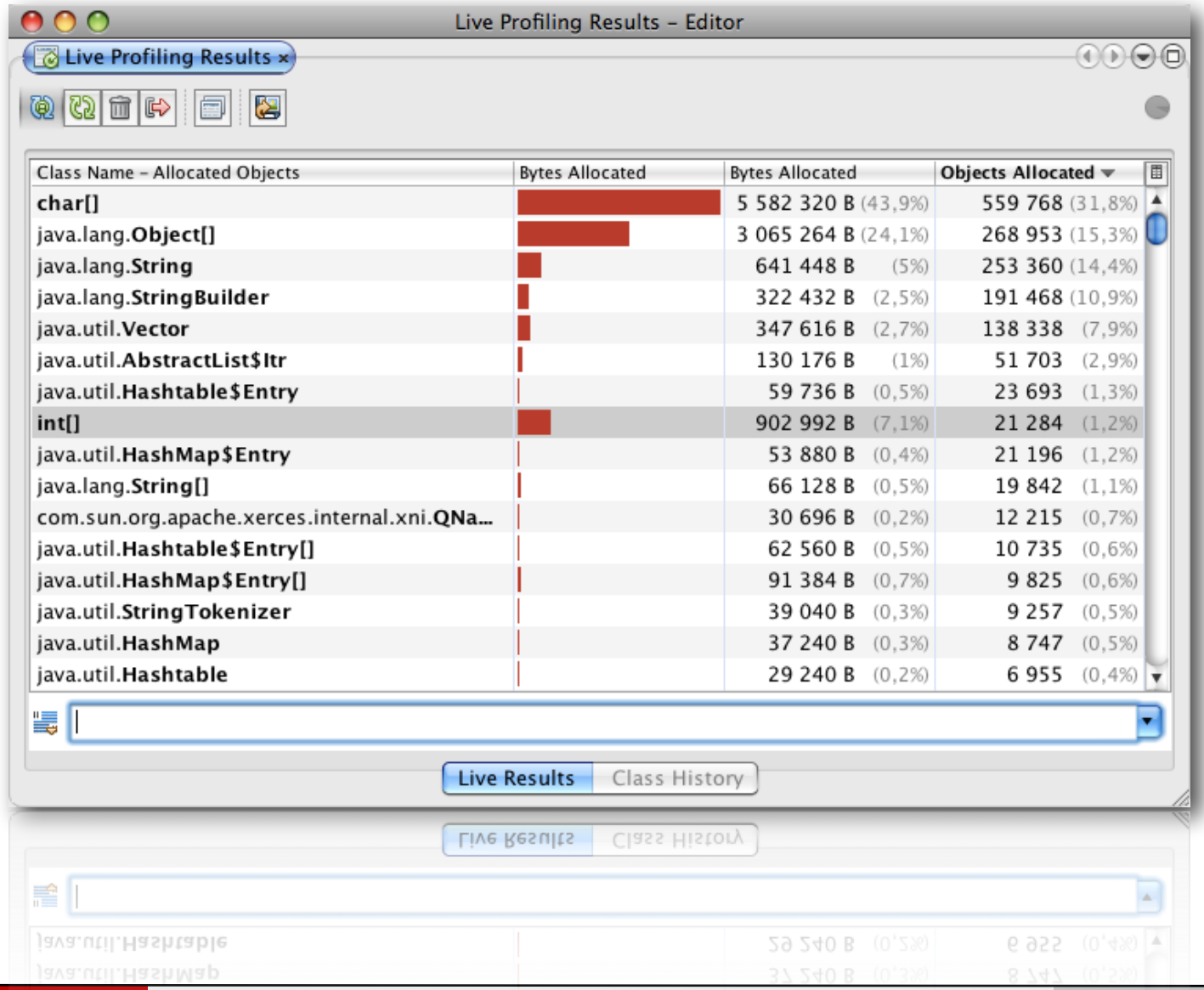
KCachegrind - une interface pour aller plus loin



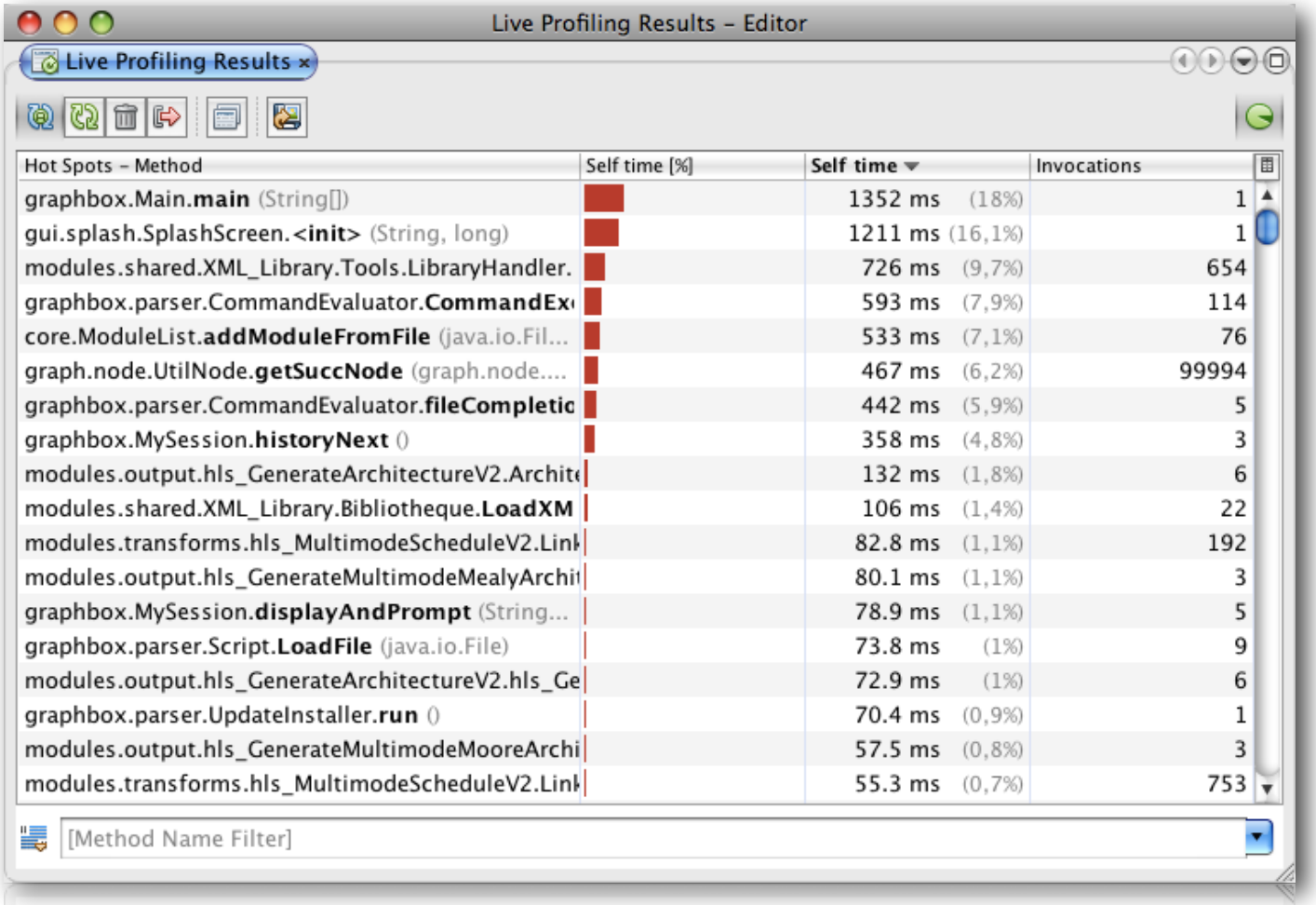
Etude de la consommation mémoire en cours d'exécution



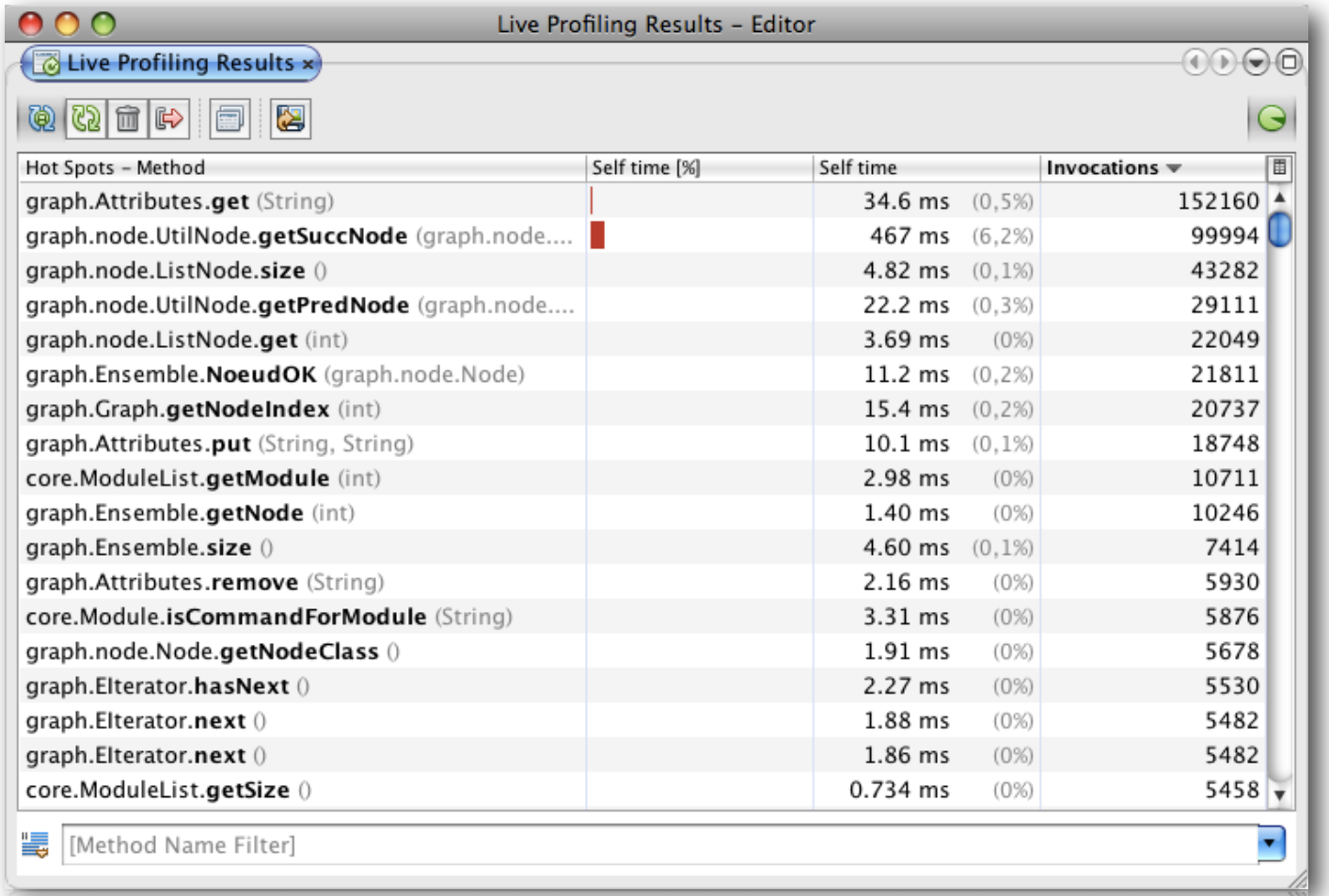
Etude de la consommation mémoire en cours d'exécution



Etude des performances temporelles post-exécution

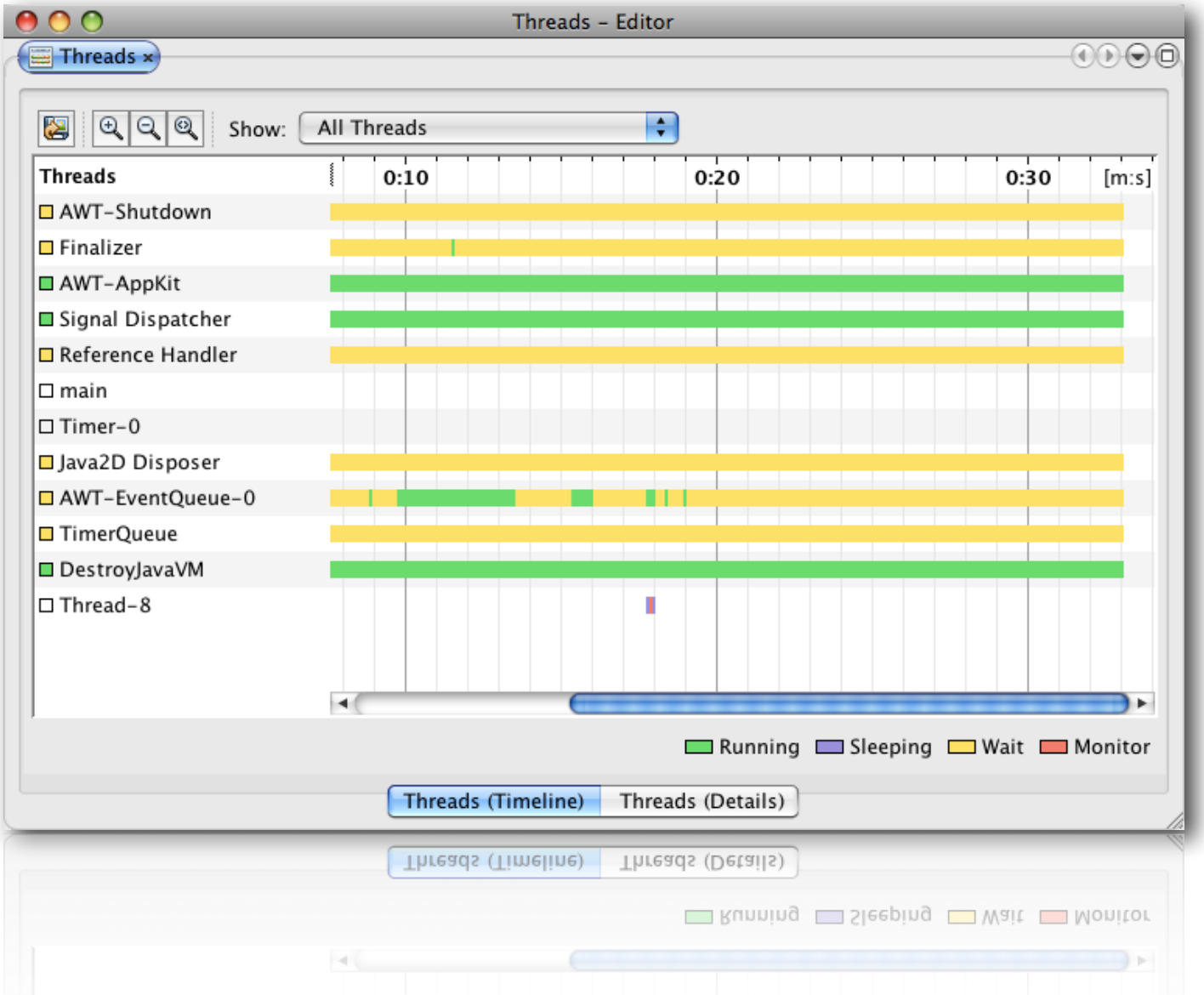


Etude des performances temporelles post-exécution



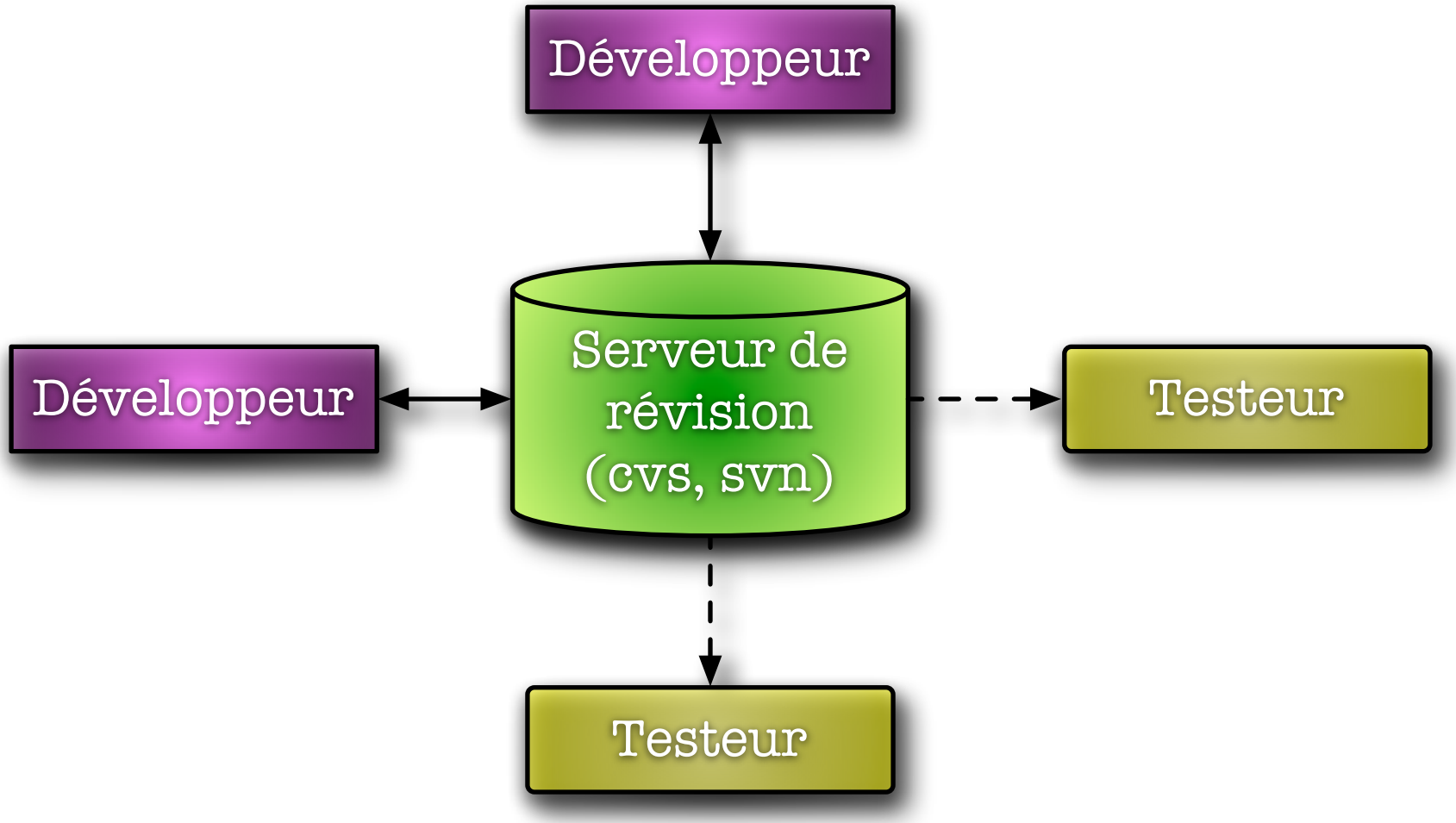
Hot Spots – Method	Self time [%]	Self time		Invocations
graph.Attributes.**get** (String)		34.6 ms	(0,5%)	152160
graph.node.UtilNode.**getSuccNode** (graph.node....		467 ms	(6,2%)	99994
graph.node.ListNode.**size** ()		4.82 ms	(0,1%)	43282
graph.node.UtilNode.**getPredNode** (graph.node....		22.2 ms	(0,3%)	29111
graph.node.ListNode.**get** (int)		3.69 ms	(0%)	22049
graph.Ensemble.**NoeudOK** (graph.node.Node)		11.2 ms	(0,2%)	21811
graph.Graph.**getNodeIndex** (int)		15.4 ms	(0,2%)	20737
graph.Attributes.**put** (String, String)		10.1 ms	(0,1%)	18748
core.ModuleList.**getModule** (int)		2.98 ms	(0%)	10711
graph.Ensemble.**getNode** (int)		1.40 ms	(0%)	10246
graph.Ensemble.**size** ()		4.60 ms	(0,1%)	7414
graph.Attributes.**remove** (String)		2.16 ms	(0%)	5930
core.Module.**isCommandForModule** (String)		3.31 ms	(0%)	5876
graph.node.Node.**getNodeClass** ()		1.91 ms	(0%)	5678
graph.ElIterator.**hasNext** ()		2.27 ms	(0%)	5530
graph.ElIterator.**next** ()		1.88 ms	(0%)	5482
graph.ElIterator.**next** ()		1.86 ms	(0%)	5482
core.ModuleList.**getSize** ()		0.734 ms	(0%)	5458

Analyse de l'exécution d'une application multi-threadée

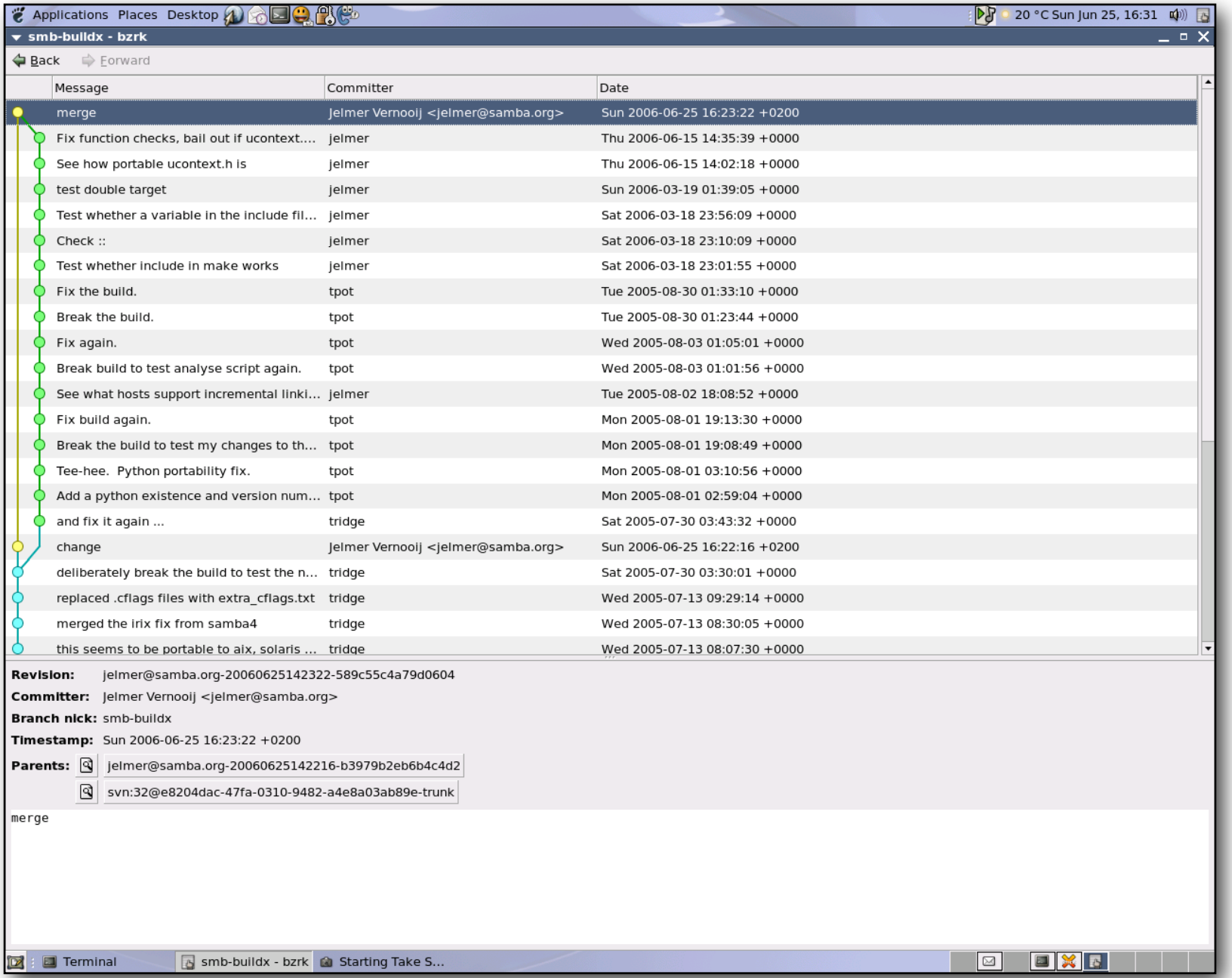


Les outils d'aide à la vérification (la gestion de projet)

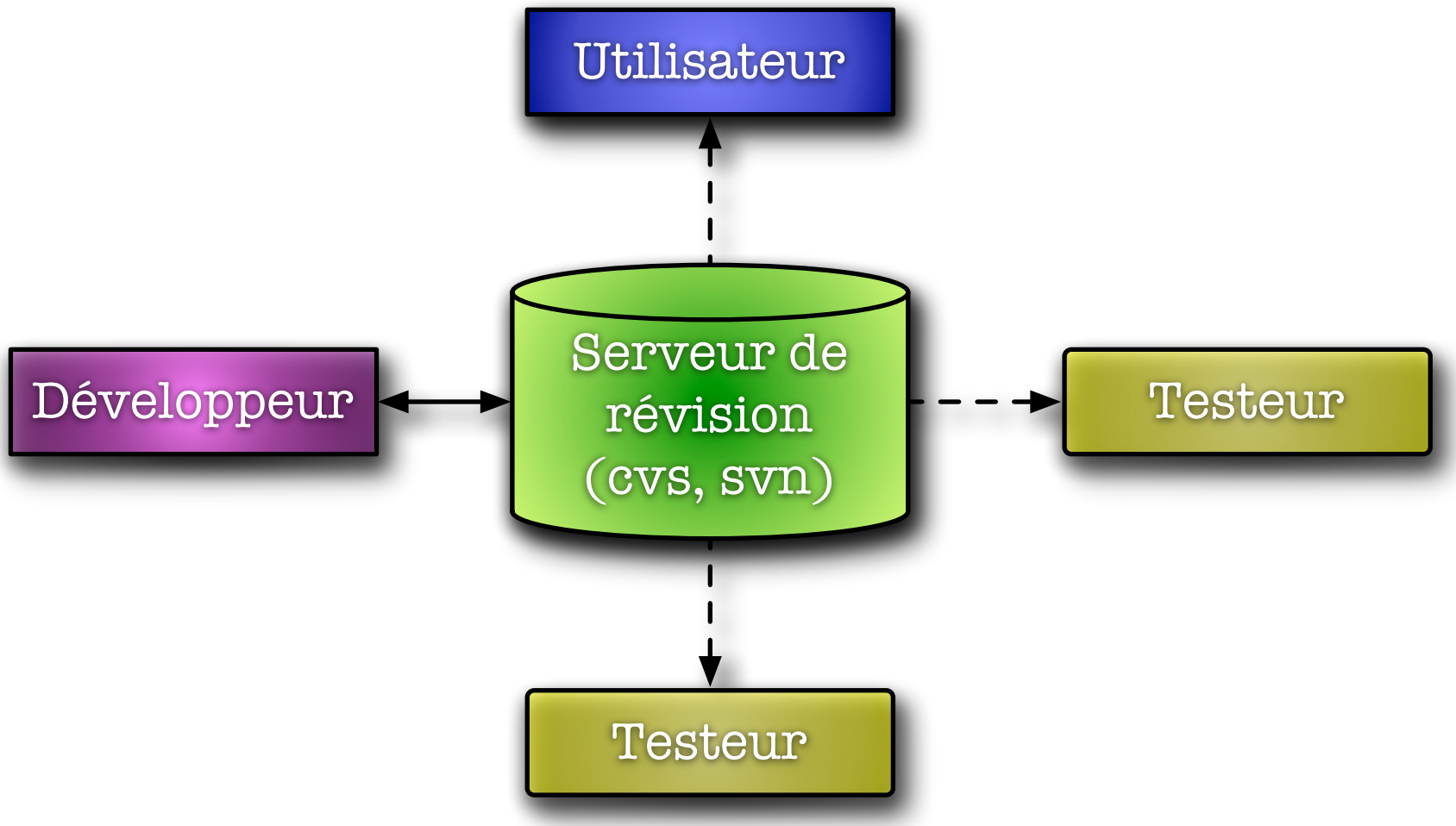
La gestion des révisions (Revision Control)



Exemple de gestion des branches



La gestion des problèmes (Issue Tracking)

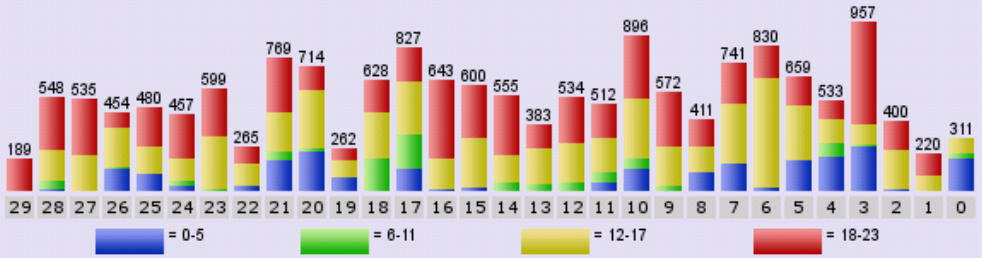


Exemple de suivi des modifications

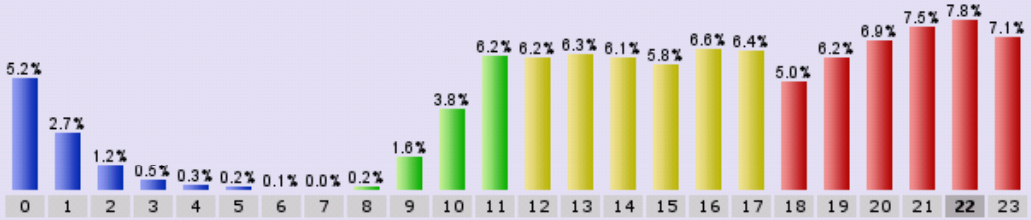
PSI Statistics

Statistics have been generated at "26.03.2006 16:56:50"

Daily activity



Most active times



Contacts

	Nick	Posts	Time	To	From
1	{nickname}	50887		43.97%	56.03%
2	{nickname}	21294		47.30%	52.70%
3	{nickname}	15906		50.67%	49.33%
4	{nickname}	11860		48.63%	51.37%
5	{nickname}	10014		48.18%	51.82%
6	{nickname}	9010		41.13%	58.87%
7	{nickname}	2783		51.96%	48.04%
8	{nickname}	2508		56.14%	43.86%
9	{nickname}	2292		31.89%	68.11%
10	{nickname}	1851		45.33%	54.67%
11	{nickname}	1792		61.83%	38.17%
12	{nickname}	1753		49.97%	50.03%
13	{nickname}	1727		49.74%	50.26%
14	{nickname}	1444		47.30%	52.70%
15	{nickname}	1110		53.51%	46.49%
16	{nickname}	1088		49.08%	50.92%

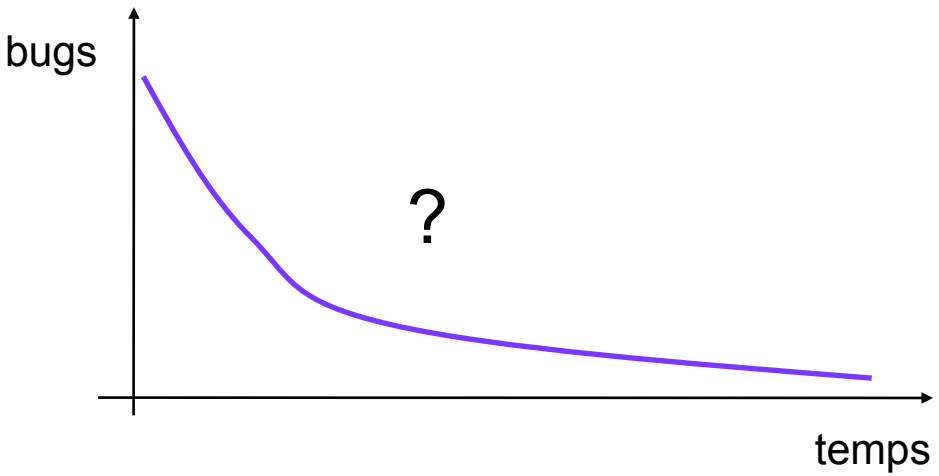
Evaluation de l'avancement de la vérification d'un système

Métriques (qualité, complexité, réutilisabilité, …)

- Management de la vérification : des métriques sont nécessaires comme point de mesure. Elles doivent permettre de représenter la situation courante du projet.
  - Où en sommes nous aujourd'hui dans le développement et la vérification de notre projet ?
  - Progresse-t-on ou régresse-t-on ?
  - Est-on loin de la fin du projet ?
  - La vérification avance-t-elle vite ou lentement ?
- Pour pouvoir répondre à ces interrogations, il faut obligatoirement posséder l'historique des métriques !

Métriques (qualité, complexité, réutilisabilité, …)

- Métriques liées au code
  - Nombre de lignes de code nécessaires pour la vérification (exclusivement),
  - Métrique permettant de connaître la capacité (efficacité) d'un langage/outil/ méthodologie à faciliter le développement d'un composant,
  - Le rapport nombre de lignes de code du composant à implémenter sur nombre de lignes de code nécessaires à sa vérification => mesure de la complexité du design,
    - Corollaire : au bout de quelques designs, à partir d'une complexité estimée d'un design, on peut prévoir l'effort qu'il faudra fournir pour sa vérification,
  - Nombre de changements dans le code en fonction du temps,
    - Cette courbe doit tendre vers 0. Monotone décroissante ?

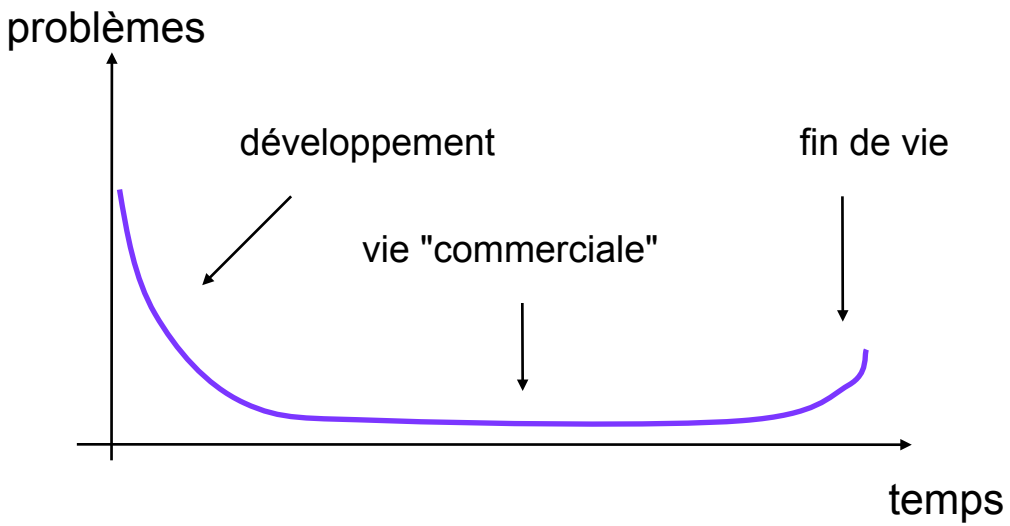


2 caps :
- début codage fonctionnalité
- début codage vérification

Permet de savoir quand le design devient stable, d'identifier les parties critiques (difficiles à concevoir) en vue de prochains designs, …

Métriques (qualité, complexité, réutilisabilité, …)

- Les métriques liées à la qualité (paramètre très subjectif…)
  - Le nombre de testbenches passés sans avoir généré d'erreur fonctionnelle, couplé au taux de couverture
  - Le nombre de problèmes détectés et en cours de correction
  - Le nombre de problèmes détectés en cours de vie du composant (critère de qualité en vue de la réutilisation du composant)



Adoptez quelques bonnes manières…

Coder avec style & rigueur (1)

- Un code peut vite devenir difficile à maintenir, à corriger, à faire évoluer car sa compréhension est difficile,
  - Son propre code quelques mois après l'avoir développé,
  - Le code d'un autre concepteur avec d'autres habitudes,
- Bon style – bonne habitudes
  - Question de discipline
  - Pas de loi empêchant de mal coder, mais mal coder conduira à une perte de temps lors d'une modification future
  - Dans des cas extrêmes, un mauvais style peut impliquer une réécriture intégrale
  - Le temps perdu maintenant sera gagné dans le support qu'il ne sera pas nécessaire d'assurer par la suite ! C'est un investissement…

Coder avec style & rigueur (2)

- Faire attention à son style d'écriture…
  - La première lettre du nom de la classe en majuscule,
  - La liste des membres publics en premier,
  - Les noms des méthodes en minuscules (tout au moins la première partie si le nom est composé),
  - Le caractère _ comme premier caractère du nom d'une donnée membre,
- Ces règles simples permettent de relire plus facilement un code, rendant sa compréhension plus facile par des personnes extérieurs.
- Des conventions de codage existent en fonction des langages que vous utilisez :
  - Sun propose un guide de bonne conduite pour le JAVA,
  - Dans certaines entreprises les concepteurs doivent suivre à la lettre les consignes du service informatique.

Coder avec style & rigueur (3)

- La première lettre du nom de la classe en majuscule,
  - class Complex
  - class Nombre
- Les noms des méthodes en minuscules (tout au moins la première partie si le nom est composé),
  - int getPixel(int x, int y)
  - char* lireNom( )
- Le caractère "_"comme premier caractère du nom d'une donnée membre,
  - void ecrireValeur(int _valeur, int _position)
  - double FIR(int _echantillons[32], int _donnees[32])

La structure des répertoires

- Utiliser des structures équivalentes pour tous les types de projets,
- Préférer les liens relatifs au liens absolus => permet de déplacer les données,
- Utiliser des « makefile » permettant de créer l'ensemble du projet,
  - ➡ Configuration environnement,
  - ➡ Compilation, archivage,…
  - ➡ Lancement des procédures de vérification, calcul du coverage,

Hiérarchie logicielle

./bin/	*les exécutables*
./spec/	*les spécifications*
./doc/	*les documentations*
./src/	*les sources*
./test/	*les testbenchs*
… … … … …	

Hiérarchie matérielle

./spec/	*les spécifications*
./doc/	*les documentations*
./src/	*les sources VHDL*
./test/	*les testbenchs*
… … … … …	

Normaliser ses entêtes de fichier

```
// FloatOpr.cpp -- A program to compute floatting point operations
//                 this model is ...
//
// Created by Bertrand Le Gal on 07/05/07.
// Copyright 2007 SuperSoC, France. All rights reserved.
//
// - Version 1.0 (date) : premiere version (auteur),
//
// - Version 1.1 (date) : revision 1 : correction du bug X (auteur)
//
// - Version 1.2 (date) : nouvelle fonctionnalite (auteur)
//
```

Il est important de conserver le nom de l'auteur ainsi que la date de création des fichiers (pour savoir sur qui taper en cas de problèmes).

De plus conserver la date et la nature des modifications apportées au codes sources est un plus pour la traçabilité des évolutions (et suivre leurs auteurs <=> évite aussi de confondre les versions)..

Les commentaires dans le code source

- La présence de commentaires est essentielle,
- De la qualité des commentaires va dépendre la réutilisation,
  - ➡ Dans certains cas, il est plus "rapide" de refaire que d'essayer de comprendre...
- Les commentaires doivent,
  - ➡ Etre clairs et concis,
  - ➡ Décrire la transformation réalisée par le code ainsi que ses E/S,
- Définir précisément les conditions d'utilisation des blocs (quelles sont les hypothèses de fonctionnement ?)

```
Commentaire inutile
/* Incrémenter la var. i */
i++;

Commentaire utile
/* Déplacer le pointeur
   l'élément d'entrée suivant */
i++;
```

Recommandations d'ordre générale

- Effacer le code faux (ne pas le commenter comme étant un faux)
  - La mémorisation de l'évolution du code source est à la charge des serveurs de révision (CVS, SVN, etc.),
  - Evite les questions inutiles,
- Mise en forme du code
  - Mettre en forme pour faciliter la lisibilité et la maintenabilité
  - Mettre une seule commande par ligne
- Limiter ls imbrications conditionnelles,
  - Difficile à vérifier (path coverage)
  - L'idéal est pas plus de 3 niveaux.

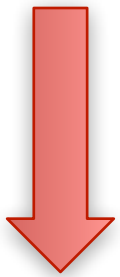
=> Pourquoi cette partie de code est fausse ? Et si je le réactivais pour résoudre mon autre problème ???

```
// Code faux
/* Ce code est faux
   i = i +2; */
   i++;

// Plusieurs commandes
if(a>b) i++; j*=2;

// Fonction max3 (illisible)
d = (a>b)?((a>c)?a:c):((b>c)?b:c);
```

=> Pourquoi ai je écrit cela ? Ce que je voulais faire est il ce que j'ai fait ???

Utilisation de fonctions (macro-préprocesseur)…

```
e = msgQLib( msg1, boite1, strlen(boite1), Normal, NULL);
f = msgQLib( msg2, boite2, strlen(boite2), Normal, NULL);
```



L'utilisation de définitions pré-processeur permet de simplifier l'écriture, d'augmenter la réutilisation, la portabilité et augmente significativement la lisibilité du code source !

```
#define SendMsg(a,b) msgQLib( a, b, strlen(b), Normal, NULL)

e = SendMsg( msg1, boite1);
f = SendMsg( msg2, boite2);
```

La gestion d'erreurs lors de l'exécution

- Toutes les erreurs pouvant avoir lieu lors de l'exécution doivent être gérées,
  - ➡ Ne pas oublier la "célèbre" loi de Murphy qui frappe régulièrement ?!
- On utilise des structures conditionnelles,
  - ➡ Affichage de messages pour informer des problèmes (warning, infos, erreurs),
  - ➡ Traitements particulier (arrêt du programme, changement de comportement, etc.)
- Ne jamais partir du principe qu'une erreur est impossible ?!

```
if( strlen(fichier) == 0 ){
    // gestion erreur
    exit(0);
}
f = fopen( fichier );
if( f == NULL ){
    // gestion erreur qui peut
    // survenir a chaque execution
}
```

```
cout << "(II) Information ..." << endl;

cout << "(WW) Le noeud n'a pas ..." << endl;

cout << "(EE) Argument incorrect" << endl;
```

Utilisation efficace du préprocesseur

- Possibilité de définir différents niveau de traces en fonction des besoins courants,
  - ➡ Activation & désactivation automatique lors de la compilation si nécessaire sans modifier le code source,
- Méthode employée pour les versions de développement,
- Les constantes dans :
  - ➡ Le code source
    #define DEBUG_LEVEL_1
  - ➡ Lors de la compilation
    gcc main.cpp -DDEBUG_LEVEL_1

```
// Fonction de division avec code de tracage
// parametrable en fonction des besoins
int division(int a, int b){
    int c = a / b;
    #ifndef DEBUG_LEVEL_2
        cout << "Diviseur  = " << a << endl;
        cout << "Dividende = " << b << endl;
    #endif;
    #ifndef DEBUG_LEVEL_1
        cout << "Resultat = " << c << endl;
    #endif;
}
```

En definisant ou non les constantes du DEBUG_LEVEL_X on active ou non les commandes d'affichage des données intermédiaires.

L'encapsulation pour se simplifier la vie

- Penser à l'encapsulation : l'encapsulation des fonctions cache les détails liés à l'implémentation (packages VHDL)
  - ➡ Permet de découpler l'usage d'une fonction de son implémentation,
  - ➡ Facilite la réutilisation de code et son évolutivité (dupliquer = baisse de la maintenabilité),
  - ➡ Permet de modifier, optimiser une fonction sans affecter le reste du code
- Point de départ de la philosophie objets (encapsulation, héritage, surcharge) !

Attention toutefois à ne pas faire "trop" de découpages => Augmentation de la taille du code pour aucun gain réel !

```
int Transfert_Reseau(String adr, String data){
    int socket = Connexion_Reseau(adr);
    Transfert_Donnee(socket, data);
    Deconnexion_Reseau(int socket);
}

int Connexion_Reseau(String adr){ /* ... ... ... */ }

void Transfert_Donnee(int socket, String data){ /* ... ... ... */ }

void Connexion_Reseau(int socket){ /* ... ... ... */ }
```