Une introduction à la vérification formelle de codes critiques

Xavier Leroy

INRIA Paris-Rocquencourt

Rencontre INRIA-industrie, 2010-05-17

Introduction

Aujourd'hui, la qualification des logiciels embarqués critiques repose principalement sur des tests rigoureux.

Que faire lorsque

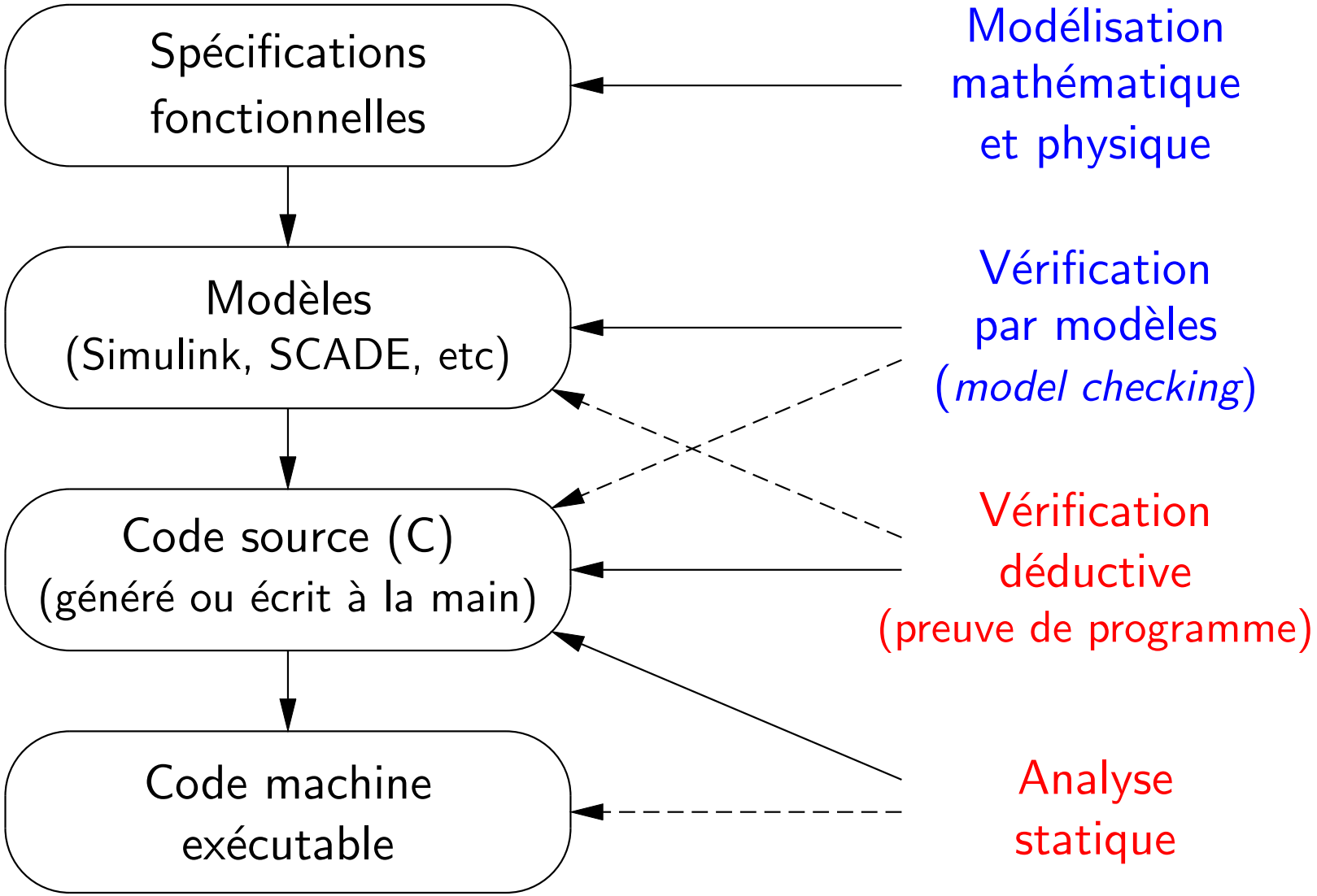
- le test ne suffit plus à garantir les exigences attendues ?
- ou devient trop coûteux à mettre en oeuvre ?

Réflexe de l'ingénieur : utiliser des mathématiques !

Réflexe de l'informaticien : se faire aider par des outils informatiques !

$\rightarrow$ les outils de vérification formelle de code.

Panorama des méthodes formelles



Cet exposé

Rapide introduction à la vérification formelle de code source et exécutable :

1. Analyse statique.
2. Vérification déductive (preuve de programmes).

Exemples d'outils de vérification.

Le problème du compilateur et des générateurs de code : préservent-ils les garanties obtenues par vérification du code source ?

Plan

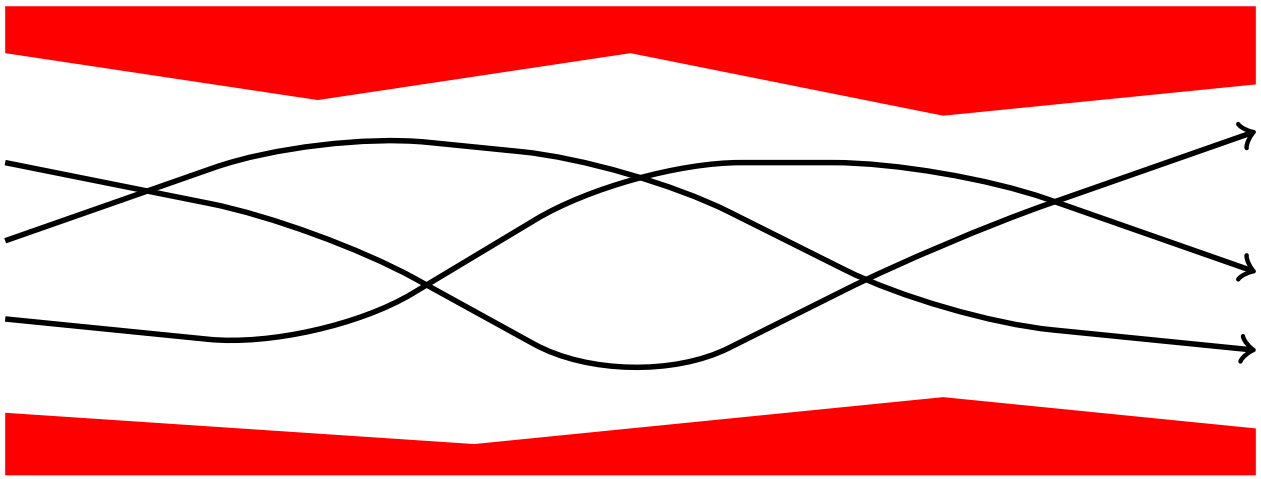
1. Analyse statique
2. Vérification déductive
3. Compilation : renforcer la confiance
4. Conclusions

L'analyse statique de code

Approximer (par un sur-ensemble) tous les comportements possibles d'un programme

- sans connaître ses entrées
- en temps fini et raisonnable
- sans exiger d'annotations sur le code.

En déduire que le programme n'a pas de comportement dangereux.

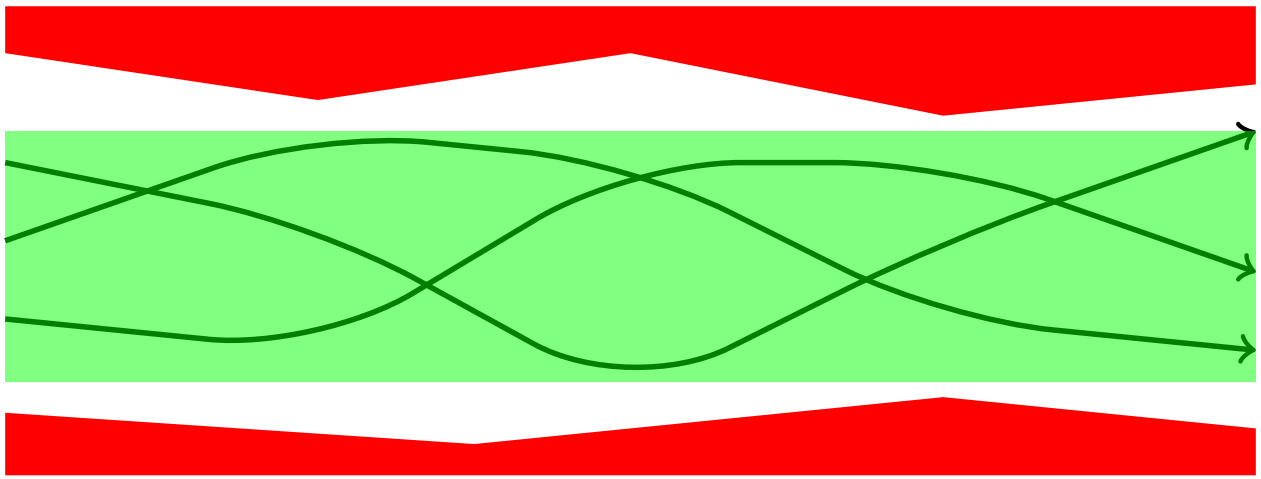


L'analyse statique de code

Approximer (par un sur-ensemble) tous les comportements possibles d'un programme

- sans connaître ses entrées
- en temps fini et raisonnable
- sans exiger d'annotations sur le code.

En déduire que le programme n'a pas de comportement dangereux.

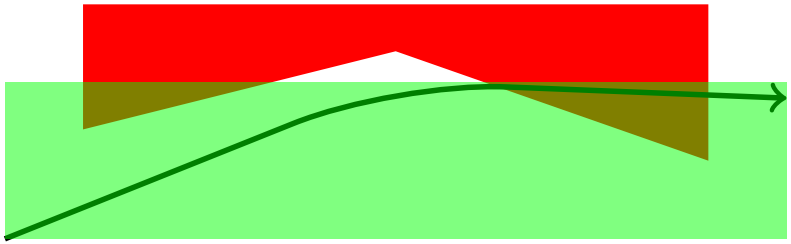


Exemple d'analyse statique

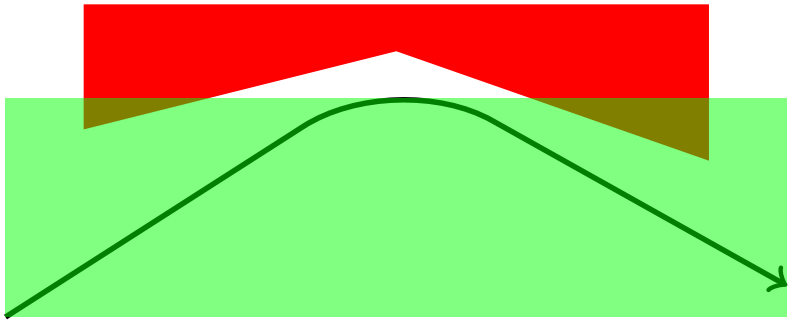
```
unsigned char input[10];
double table[256];
int i, somme, moyenne;

somme = 0;
for (i = 0; i < 10; i++) {
    somme += input[i];              // somme ∈ [0, 2550]
}
moyenne = somme / 10;               // moyenne ∈ [0, 255]
return table[moyenne];              // accès dans les bornes
```

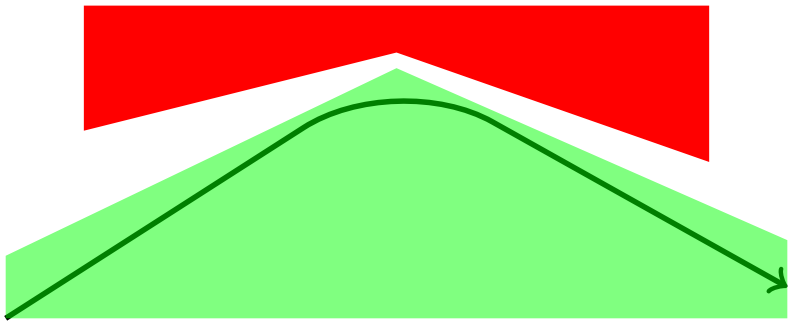
Vraies et fausses alarmes



Vraie alarme
(comportement dangereux)



Fausse alarme
(analyse pas assez précise)



Approximation plus fine (polyèdre au lieu de rectangle) :
pas d'alarme

Quelques propriétés garanties par analyse statique

Absence d'erreurs à l'exécution :

- Tableaux et pointeurs :
  - Pas d'accès hors-bornes.
  - Pas de déréférencement du pointeur nul.
  - Pas d'accès après un `free`.
  - Contraintes d'alignement du processeur.
- Entiers :
  - Pas de division par zéro.
  - Pas de débordements arithmétiques.
- Flottants :
  - Pas de débordements arithmétiques (infinis).
  - Pas d'opérations indéfinies (not-a-number).

Intervalles de variation des sorties du programme.

Exemple d'analyse de calculs flottants

Prise en compte des arrondis et de leur propagation.

```
float x, y, u, v;          // x ∈ [1.00025, 2]
                           // y ∈ [0.5, 1]
u = 1 / (x - y);           // OK
v = 1 / (x*x - y*y);       // ALARME: résultat indéfini
```

Dans le premier cas, $(x - y) \in [0.00025, 1.5]$ et la division ne peut pas produire un infini ou un not-a-number.
Dans le second cas,

$$\begin{array}{rcll} (x*x) & \in & [1, 4] & \text{(arrondi flottant !)} \\ (y*y) & \in & [0.25, 1] & \\ (x*x - y*y) & \in & [0, 3.75] & \end{array}$$

et on peut diviser par zéro.

Différents types d'analyses statiques

Analyses de valeurs : propriétés d'une seule variable.

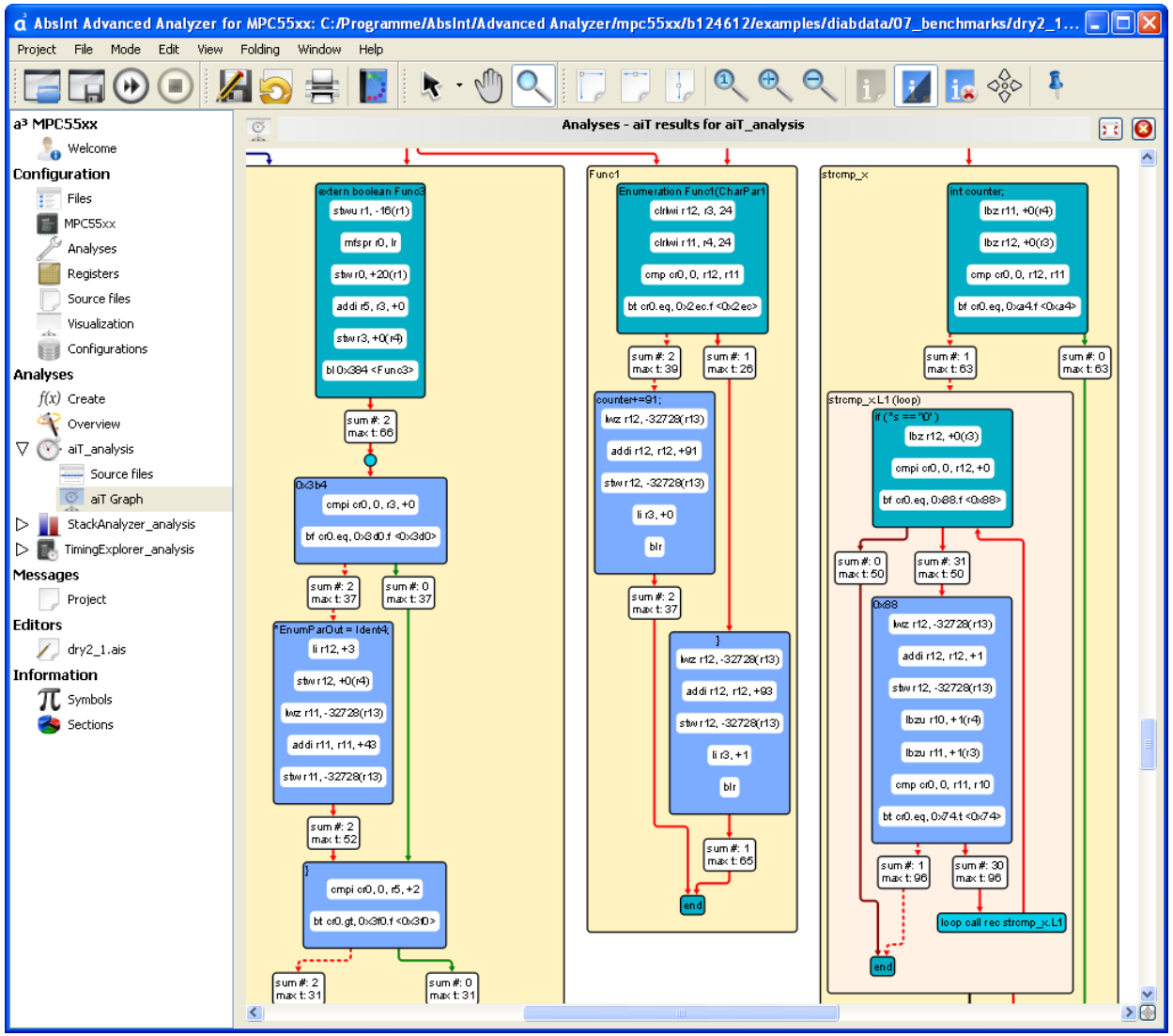
- De type numérique : intervalle de variation $x \in [a, b]$.
- De type pointeur : validité des accès mémoire, non-aliasing.

Analyses relationnelles : invariants entre plusieurs variables.
(Exemple : polyèdres = inégalités linéaires $ax + by \leq c$.)

Analyses de propriétés «non fonctionnelles» :

- Consommation mémoire.
- Temps d'exécution (WCET).

Analyse de temps d'exécution : aiT WCET



Quelques outils d'analyse statique

Outils «généralistes» :

- Coverity
- MathWorks Polyspace verifier.
- Frama-C value analyzer.

Outils spécialisés à un domaine d'application :

- Microsoft Static Driver Verifier (code système Windows)
- Astrée (codes de contrôle-commande).
- Fluctuat (analyse symbolique des erreurs en flottants).

Outils opérant sur le code machine après compilation :

- aiT WCET, aiT StackAnalyzer.

Plan

1. Analyse statique
2. Vérification déductive
3. Compilation : renforcer la confiance
4. Conclusions

Vérification déductive (preuve de programmes)

Annoter le programme avec des formules logiques :

- Préconditions (exigences sur les arguments de fonctions)
- Postconditions (garanties sur les résultats de fonctions)
- Invariants de boucles.

Vérifier que :

- Pour chaque fonction, préconditions impliquent postconditions.
- Pour chaque appel de fonction, les préconditions sont satisfaites.

Outillage :

- Générateurs d'obligations de vérifications.
- Démonstrateurs automatiques.

Exemple de vérification déductive

Recherche dichotomique dans une table.

```
int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;
  while (l <= u) {
    int m = l + (u - l) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else return m;
  }
  return -1;
}
```

Spécification des pré- et post-conditions

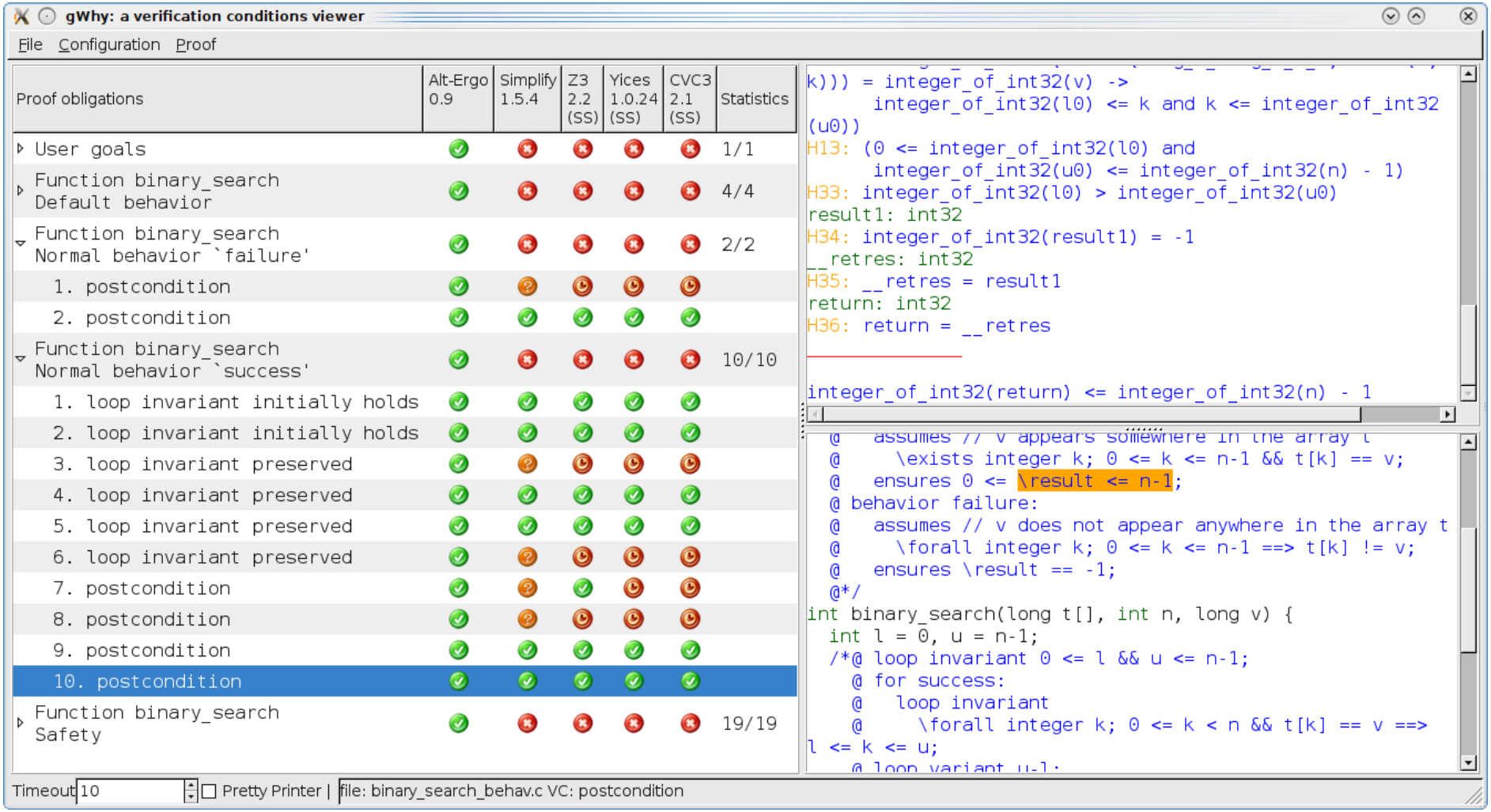
Dans le langage ACSL de l'outil Frama-C :

```
/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ behavior success:
  @   assumes // array t is sorted in increasing order
  @     \forall integer k1, k2;
  @              0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @   assumes // v appears somewhere in the array t
  @     \exists integer k; 0 <= k <= n-1 && t[k] == v;
  @   ensures 0 <= \result <= n-1 && t[\result] == v;
  @ behavior failure:
  @   assumes // v does not appear anywhere in the array t
  @     \forall integer k; 0 <= k <= n-1 ==> t[k] != v;
  @   ensures \result == -1;
  @*/
```

Spécification de l'invariant de boucle

```
int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;
  /*@ loop invariant 0 <= l && u <= n-1;
    @ for success:
    @   loop invariant
    @     \forall integer k;
    @         0 <= k < n && t[k] == v ==> l <= k <= u;
    @ loop variant u-l;
    @*/
  while (l <= u) {
    int m = l + (u - l) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else return m;
  }
  return -1;
}
```

Production et preuve des obligations



Comparaison analyse statique / vérification déductive

	Analyse statique	Vérification déductive
Propriétés garanties	Absence d'erreurs à l'exécution	Propriétés fonctionnelles arbitrairement complexes
Annotations manuelles	Pas ou très peu	Beaucoup
Passage à l'échelle	10^5–10^6 lignes de code	10^2–10^3 lignes de code

Outils et exemples

Quelques outils de vérification déductive :

- Pour Java : ESC/Java
- Pour C# : Spec# / Boogie
- Pour C : Caveat, Frama-C / Jessie

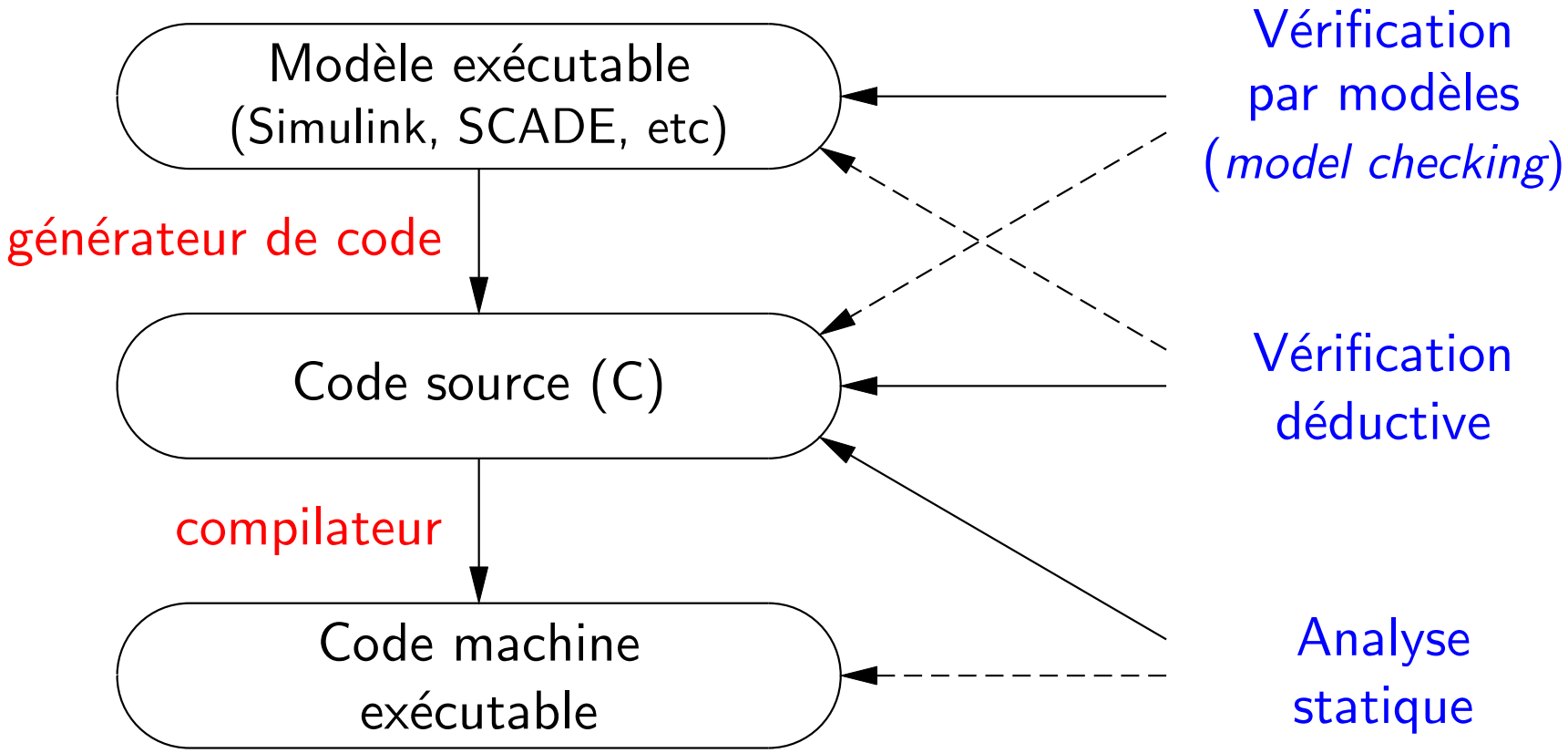
Quelques exemples de vérifications de grande taille :

- L4.verified (NICTA, Australie) :
  vérification du micro-noyau sécurisé seL4 (8000 lignes).
- Verisoft (Allemagne) :
  vérification complète d'un composant automobile
  (du circuit jusqu'au code applicatif).

Plan

1. Analyse statique
2. Vérification déductive
3. Compilation : renforcer la confiance
4. Conclusions

Compilateurs et générateurs de code



Les garanties obtenues par vérification formelle sur le modèle ou sur le source C s'étendent-t'elles au code machine exécutable ?

Pourquoi la compilation pose problème ?

Les compilateurs et générateurs de code sont des programmes complexes qui effectuent des transformations de code non triviales :

- Le ≪fossé sémantique≫ qui sépare langage source et code machine.
- Amélioration des performances du code produit (optimisation).

Un bug dans le compilateur peut faire produire un exécutable faux à partir d'un source correct.

Un exemple de compilation optimisée

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compilé avec le compilateur Tru64/Unix et retranscrit manuellement en C...

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

```
      if (4 >= r2) goto L14;
      prefetch(a[20]); prefetch(b[20]);
      f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
      r1 = 8; if (8 >= r2) goto L16;
L17:  f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
      f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
      f12 = a[4]; f16 = f18 * f16;
      f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
      f11 += f17; r1 += 4; f10 += f15;
      f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
      f1 += f16; dp += f19; b += 4;
      if (r1 < r2) goto L17;
L16:  f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
      f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
      f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
      a += 4; b += 4; f14 = a[8]; f15 = b[8];
      f11 += f22; f1 += f21; dp += f24;
L18:  f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
      f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
      a += 4; f28 = f29 * f28; b += 4;
      f10 += f14; f11 += f12; f1 += f26;
```

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

Un exemple de compilation non optimisée mais délicate

```
double floatofint(unsigned int i) { return (double) i; }
```

L'architecture PowerPC 32 bits ne fournit pas d'instruction de conversion entier $\rightarrow$ flottant. Le compilateur doit donc l'émuler, comme suit :

```
double floatofint(unsigned int i)
{
    union { double d; unsigned int x[2]; } u, v;
    u.x[0] = 0x43300000;   u.x[1] = i;
    v.x[0] = 0x43300000;   v.x[1] = 0;
    return u.d - v.d;
}
```

(Indication : l'entier 64 bits $\texttt{0x43300000} \times 2^{32} + x$ est le codage IEEE754 du flottant double précision $2^{52} + (\texttt{double})x$.)

Les bugs dans les compilateurs

Il est établi que les compilateurs du commerce compilent «de travers» de nombreux codes sources :

> *NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.*
>
> `http://www.nullstone.com/htmls/category/divide.htm`

> *We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables. This result is disturbing because it implies that embedded software and operating systems — both typically coded in C, both being bases for many mission-critical and safety-critical applications, and both relying on the correct translation of volatiles — may be being miscompiled.*
>
> *E. Eide & J. Regehr, EMSOFT 2008*

Que faire ?

Approche classique :

- Utiliser des compilateurs ≪qualifiés par l'usage≫.
- Désactiver toute les optimisations.
- Encore plus de tests.
- Revues manuelles du code assembleur produit.

Approche formelle : vérifier formellement le compilateur lui-même pour prouver un résultat de préservation sémantique :

> *Si le compilateur ne signale pas d'erreur à la compilation*
> *et si le code source ne contient pas d'erreurs à l'exécution,*
> *alors l'exécutable produit par le compilateur se comporte*
> *exactement comme prescrit par la sémantique du code source.*

Le projet CompCert

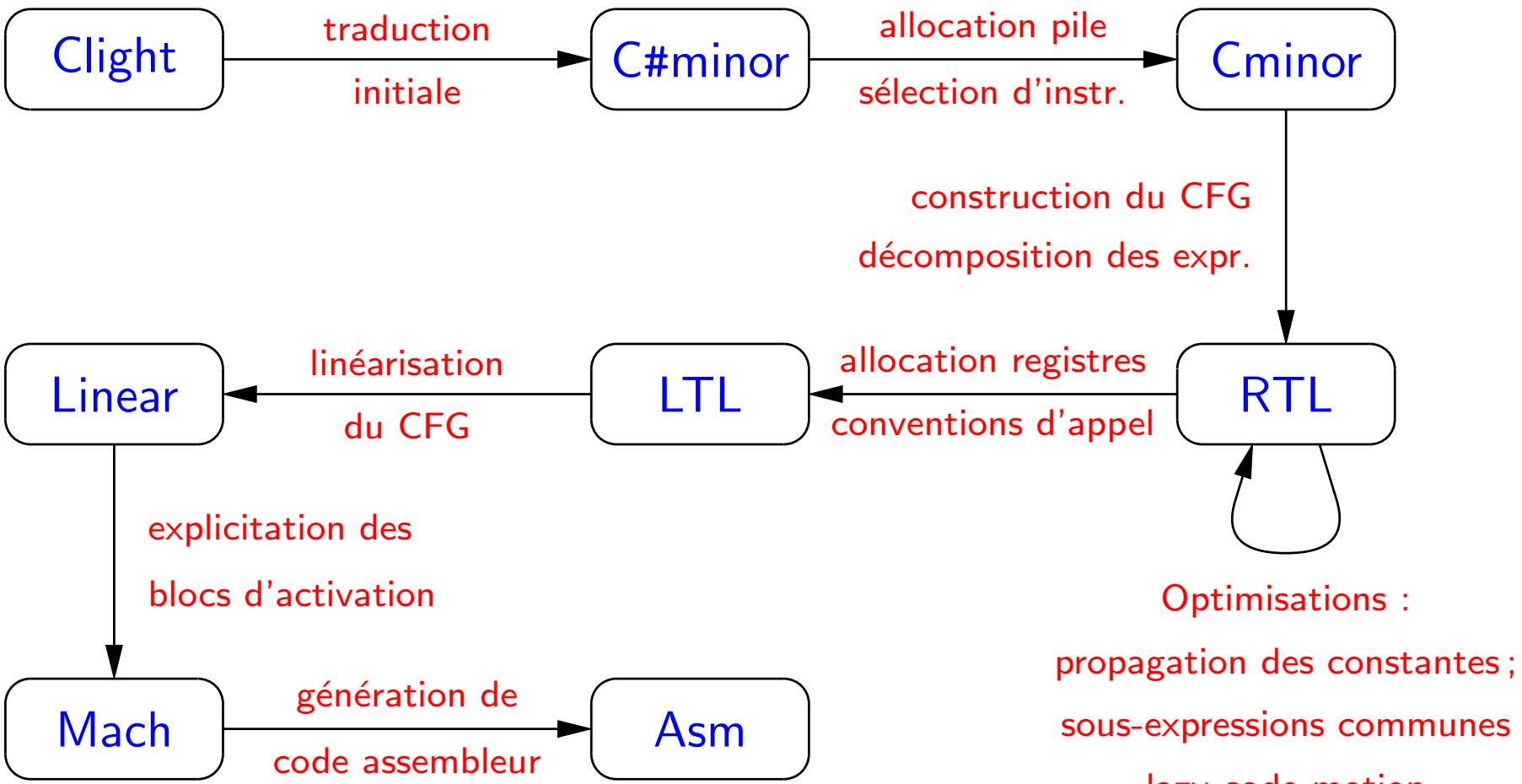
(INRIA/CNAM/Paris 7 ; X. Leroy, S. Blazy, et al)

Développement et vérification formelle d'un compilateur réaliste, utilisable pour le logiciel embarqué critique :

- Langage source : un grand sous-ensemble de C.
- Langage cible : assembleur PowerPC et ARM.
- Performances du code produit : comparables à `gcc -O1`.

La vérification (déductive) utilise l'assistant de preuve Coq.
(50000 lignes, 3 hommes-années.)

Diagramme du compilateur CompCert C



Le GAC Gene-Auto

(IRIT ; N. Izerrouken, X. Thirioux, M. Pantel, M. Strecker)

Un générateur automatique de code Simulink $\rightarrow$ C.

Vérification formelle en cours, utilisant l'assistant de preuve Coq.

Début de réflexion sur la qualification DO-178 d'un tel outil vérifié.

Plan

1. Analyse statique
2. Vérification déductive
3. Compilation : renforcer la confiance
4. Conclusions

Conclusions

Des fondations théoriques déjà anciennes . . .

(preuve de programmes : 1969 ; analyse statique : 1977 ; compilateurs vérifiés : 1972)

. . . qui sont devenues praticables ces 10 dernières années.

Une première génération d'outils disponible . . .

. . . qui commencent à être utilisés dans l'industrie du logiciel critique.

Un domaine où l'industrie aéronautique européenne est en pointe. . .

. . . et où la recherche académique est très active.

Quelques directions de travail

Prise en compte du parallélisme à mémoire partagée :

- Analyse statique (notamment du WCET).
- Vérification déductive (logiques de séparation).
- Compilation (quelles optimisations restent valides ?).

Vérification déductive fine des calculs flottants.

Vers une vérification formelle des outils de vérification.