Application : analyse de logiciels embarqués avioniques critiques

TAS : Typage et analyse statique
M2, Master STL INSTA, UPMC

Antoine Miné

Année 2016–2017

Cours 14
23 mars 2017

Plan

- **Introduction**
- **L'analyseur Astrée** de programmes **synchrones**
  - conception d'un analyseur spécialisé
  - exemples d'abstractions
  - résultats expérimentaux
- **L'analyseur AstréeA** de programmes **concurrents**
  - analyse modulaire thread-à-thread avec interférences
  - exemples d'abstractions
  - résultats expérimentaux

Introduction

Vérification des logiciels avioniques

Les logiciels avioniques critiques doivent être certifiés :

- obligation légale
- processus régulé par des standards internationaux (DO-178B, DO-178C)
- plus de la moitié des coûts de développement
- essentiellement basé sur des campagnes de tests massives
  & sur la relecture de code par des pairs (humains)

Tendance :

l'utilisation de **méthodes formelles** est officiellement reconnue
(DO-178C, DO-333)

- au niveau binaire, pour remplacer le test
- au niveau du source, pour remplacer la relecture de code
- au niveau du source, pour remplacer le test
  si la correspondance entre source et binaire est aussi certifiée
- pour vérifier la robustesse et l'absence d'erreur à l'exécution

$\Longrightarrow$ les méthodes formelles permettent d'améliorer l'efficacité et de réduire le coût de la certification !

Méthodes formelles chez Airbus

Preuve de programme : méthodes déductives

- propriétés fonctionnelles pour des petits morceaux de code séquentiel en C
- remplace le teste unitaire
- pas entièrement automatisé
- outil **Caveat** (CEA, voir aussi FramaC)

Analyse statique sûre :

- analyse entièrement automatique de grosses applications, pour des propriétés non fonctionnelles
- temps d'exécution maximal et utilisation maximale de pile, sur du binaire : **aiT**, **StackAnalyzer** (AbsInt)
- absence d'erreur à l'exécution, sur du code C séquentiel
  (dépassement de capacité, erreur arithmétique, dépassement de tableau, etc.)
  analyseur **Astrée** (AbsInt)

Compilation certifiée :

- permet l'analyse au niveau du source de certifier aussi le code binaire (séquentiel)
- compilateur C **CompCert**, certifié en **Coq** (INRIA)

Analyse statique sûre

Avantages :

- analyse directe du code source (pas un modèle séparé)
- automatique (facile à mettre en œuvre par un utilisateur, peu d'interaction nécessaire)
- efficace
- approximée (pour contourner les problème de décidabilité et de performance)

Sûreté :

- basé sur la sémantique (spécification C, entiers machines, flottants, pointeurs, . . .)
- couverture totale du contrôle et des données
- tout propriété démontrée par l'analyse est vraie de toute exécution
  $\Longrightarrow$ aucune erreur n'est oubliée : pas de faux négatif
- la sûreté est **imposée** par le standard DO-178

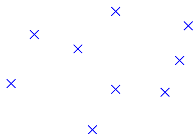
Interprétation abstraite :

théorie de l'approximation des sémantiques, permet la conception d'analyses statiques sûres avec un contrôle fin entre coût et précision

Conception “classique” d’un interprète abstrait

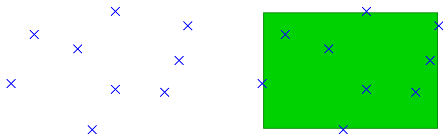
1. Écrire les règles de la sémantique concrète
2. Choisir les classes de propriétés d’intérêt
3. Déterminer la classe des propriétés qui doivent effectivement être inférées
4. Définir un analyseur sur une sémantique abstraite calculable

Conception "classique" d'un interprète abstrait



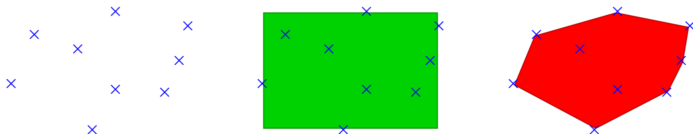
1. Écrire les règles de la sémantique concrète
   - fonction des programmes vers un monde mathématique riche
   - formalisation fidèle de la spécification du langage
   - vérité de base, sur laquelle la sûreté de l'analyse repose
   - non calculable !
2. Choisir les classes de propriétés d'intérêt
3. Déterminer la classe des propriétés qui doivent effectivement être inférées
4. Définir un analyseur sur une sémantique abstraite calculable

Conception “classique” d’un interprète abstrait



1. Écrire les règles de la sémantique concrète
2. Choisir les classes de propriétés d’intérêt
   - e.g. : bornes sur les variables, $X \in [a, b]$
3. Déterminer la classe des propriétés qui doivent effectivement être inférées
4. Définir un analyseur sur une sémantique abstraite calculable

Conception “classique” d’un interprète abstrait



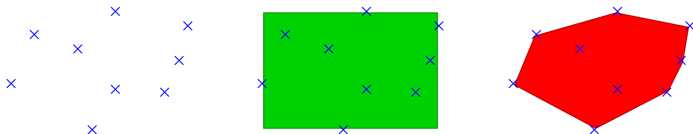
1. Écrire les règles de la sémantique concrète

2. Choisir les classes de propriétés d’intérêt

3. Déterminer la classe des propriétés qui doivent effectivement être inférées
   - généralement, plus expressives que les propriétés d’intérêt
     il faut représenter des invariants intermédiaires en tout point de programme, des invariants de boucle, etc.
   - peut dépendre de la classe de programmes analysés
   - e.g. : contraintes linéaires, $\alpha X + \beta Y \leq \gamma$

4. Définir un analyseur sur une sémantique abstraite calculable

Conception "classique" d'un interprète abstrait



1. Écrire les règles de la sémantique concrète
2. Choisir les classes de propriétés d'intérêt
3. Déterminer la classe des propriétés qui doivent effectivement être inférées
4. Définir un analyseur sur une sémantique abstraite calculable
   - dériver ou inventer des opérateurs abstraits (e.g. : arithmétique d'intervalle)
   - inventer des opérateurs d'accélération $\triangledown$
   - domaine abstrait : structures de données et algorithmes

   les calculs abstraits et $\triangledown$ accumulent des imprécisions
   $\Longrightarrow$ nous ne trouverons pas la propriété la plus précise exprimable dans l'abstrait...

L'analyseur statique Astrée

L'analyseur statique Astrée

Analyseur statique de programmes temps-réels embarqués

- développe à l'ENS (2001–2009)
  B. Blanchet, P. Cousot, R. Cousot, J. Feret,
  L. Mauborgne, D. Monniaux, A. Miné, X. Rival
- industrialisé et commercialisé par AbsInt
  (depuis 2009)

Astrée
www.astree.ens.fr

AbsInt
www.absint.com

L'analyseur statique Astrée

Analyseur statique spécialisé

Principe :

1. Partir d'un analyseur simple, rapide, peu précis (e.g., intervalles) et d'un programme caractéristique dans la classe d'intérêt

Analyseur statique spécialisé

Principe :

1. Partir d'un analyseur simple, rapide, peu précis (e.g., intervalles) et d'un programme caractéristique dans la classe d'intérêt
2. Raffiner manuellement l'analyseur jusqu'à atteindre 0 fausse alarme
   - déterminer quelles propriétés intermédiaires ne sont pas inférées
   - ajouter un nouveau domaine abstrait

     si la propriété n'est pas déjà exprimable

     utilisation d'opérateurs abstraits rapides et minimalistes, si possible
     limiter le périmètre d'activation du domaine (variables, portions de programme)
     pour rester efficace
   - raffiner des opérateurs abstraits dans des domaines existants
   - ajouter des réductions entre domaines existants
   - affiner les paramètres de précision

     périmètre d'activation, paramètres d'itération, . . .

     (ceci peut être fait par l'utilisateur de l'analyseur)

Analyseur statique spécialisé

Résultat

- sûr par construction
- efficace par parcimonie
- 0 fausse alarme sur le programme cible
- encourage une conception modulaire
  et des abstractions réutilisables

Justification théorique :

- Pour chaque programme et propriété, un domaine adéquat existe
  mais sa construction effective n'est pas mécanisable
- Un domaine donné fonctionne sur un nombre infini de programmes
- Toute combinaison finie de domaines échoue sur un nombre infini de programmes

En pratique, un analyseur sera précis sur toute une classe de programmes
La suppression des fausses alarmes au cas par cas nécessite un affinage des paramètres de précision (qui peut souvent être effectué par l'utilisateur)

L'analyseur spécialisé Astrée

Spécialisé pour :

- l'analyse des erreurs à l'exécution
  dépassements arithmétiques, dépassements de tableaux, divisions par 0, etc.
- les logiciels C embarqués critiques
  pas d'allocation dynamique de mémoire, pas de récursivité
- et en particulier les logiciels de contrôle/commande
  programmes réactifs, avec des calculs flottants intensifs
- la validation
  toutes les erreurs sont trouvées, et peu de fausses alarmes

Environ 40 domaines abstraits sont utilisés simultanément

Logiciels de contrôle/commande synchrones

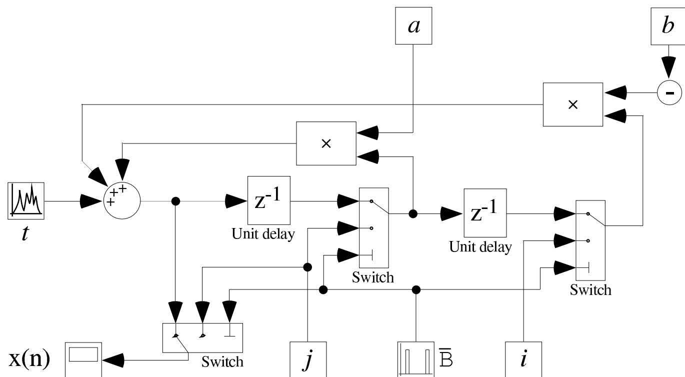
Code réactif conçu dans un langage graphique, puis compilé en C
(Scade, Simulink)

Structure

initialisation des variables d'état
`while ( clock` $\leq$ `3 600 000 ) {`
    lire les entrées des senseurs (`volatile`)
    calculer les sorties et le nouvel état
    écrire les sorties dans les actuateurs
    attendre le prochain tick d'horloge
`}`

- les structures de données restent simples (tableaux)
- l'espace d'état global est très important ($\simeq 10K$ variables)
- nombreux calculs numériques (interpolations, filtrage digital)
- la structure de contrôle est très plate
  (l'inlining de appels de fonctions est possible)

Diagramme de bloc



- chaque bloc est une (petite) fonction C prédéfinie
- une boîte a des variables d'état rémanentes (`static`)
- les entrées sont volatiles,
  uniquement bornées par des contraintes physiques

Sémantique concrète d'Astrée

Sémantique concrète : définie par

- la norme C99 (programmes portables)
- la norme IEEE 754-1985 (calculs en virgule flottante)
- paramètres spécifiques à chaque architecture (`sizeof`, endianess, `struct`, etc.)
- paramètres du compilateur et du linker (initialisation, etc.)

Propriétés d'intérêt : absence d'erreur à l'exécution

- pas de dépassement capacité en entier ni en flottant
- pas d'opération arithmétique invalide (/0, << 33)
- pas d'accès de tableau ou par pointeur invalide (tableaux [], pointeurs *)
- respect des assertions introduites dans le programme par le programmeur (`assert`)

i.e., l'**accessibilité d'un état de programme invalide**

Sémantique après une erreur

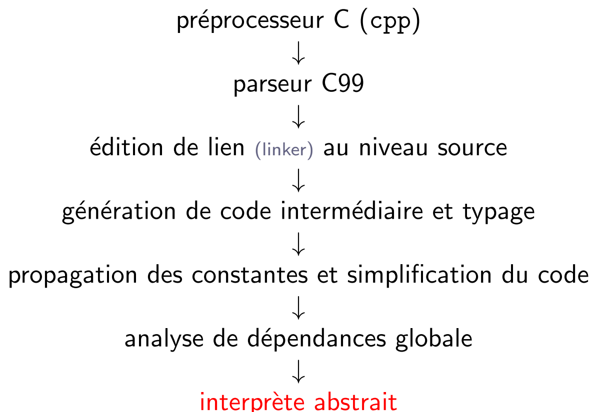
Plusieurs sémantiques sont envisageables après une erreur :

- arrêter le programme (i.e., l'opérateur renvoie $\perp$)
  - division ou modulo par zéro
  - dépassement de capacité en flottant (selon la configuration du FPU)
  - échec d'un `assert`
- retourner une valeur arbitraire
  (non-déterminisme dans l'ensemble des valeurs autorisées par le type)
  - décalage de bit invalide
- résultat bien défini
  - arithmétique modulo 2^n en entiers non-signés
  - *type-punning*
    (réinterprétation d'un motif de bits comme une valeur d'un autre type)
- comportement totalement indéfini (traité comme un arrêt de programme)
  - accès mémoire invalide

Certains comportements peuvent être paramétrés par l'utilisateur.

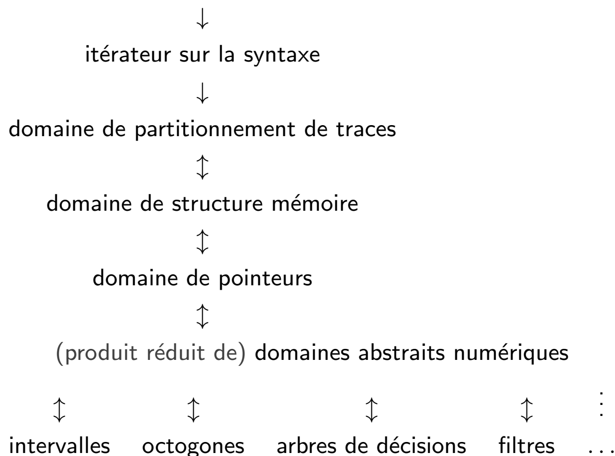
Il est important de continuer l'analyse après une alarme !

Vue générale de l'analyseur

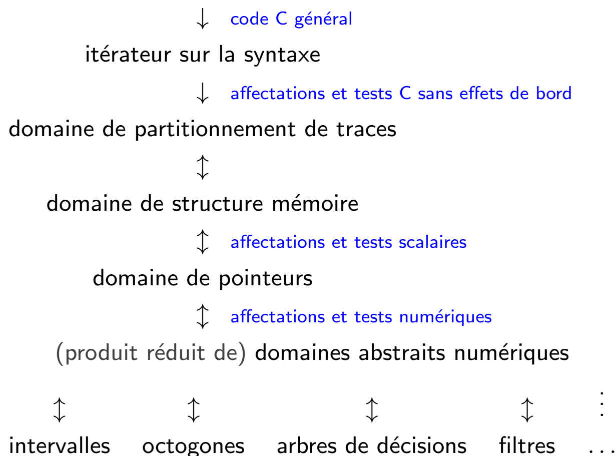


nombreux points communs avec le *font-end* d'un compilateur

Interprète abstrait d'Astrée



Interprète abstrait d'Astrée



Interprète abstrait d'Astrée

↓ `for (i=0;...) a[i] = *p;`

itérateur sur la syntaxe

↓ `a[i] = *p`

domaine de partitionnement de traces

↕ `a[0] = *p, a[1] = *p, ...`

domaine de structure mémoire

↕ `a@0 = x`

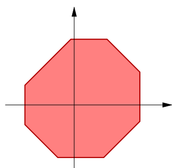
domaine de pointeurs

↕ `a@0 = x`

(produit réduit de) domaines abstraits numériques

↕	↕	↕	↕	⋮	`a@0 = x`
intervalles	octogones	arbres de décisions	filtres	...	

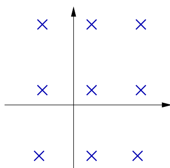
Quelques domaines abstraits utilisés dans Astrée



octogones

$\pm X \pm Y \leq c$

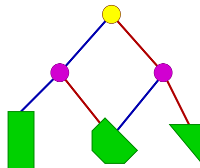
[Miné 2006]



congruences

$X \equiv a\,[b]$

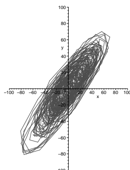
[Granger 1989]



arbres de décision booléens

disjonctions

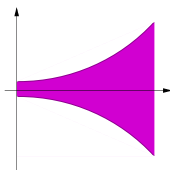
[Mauborgne]



ellipsoïdes

filtres digitaux

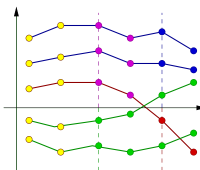
[Feret 2005]



exponentielles

$X \leq (1+\alpha)^{\beta t}$

[Feret 2005]



partitionnement de traces

disjonctions

[Mauborgne Rival 2005]

Exemple : paquets d'octogones

Invariants : $\wedge_{ij} \pm X_i \pm X_j \leq c_{ij}$; coût cubique

Un coût $(|\mathbb{V}|^n)$ avec $n > 1$ est trop élevé en pratique

Exemple : paquets d'octogones

Invariants : $\wedge_{ij} \pm X_i \pm X_j \leq c_{ij}$; coût cubique

Un coût ($|\mathbb{V}|^n$) avec $n > 1$ est trop élevé en pratique

Solution

Ne pas mettre toutes les variables $\mathbb{V}$ dans un unique octogone,
créer à la place de nombreux petits "paquets" d'octogones :

- déterminés par une pré-analyse de dépendance
- ne relier dans un octogone que les variables modifiées ensemble
- interrompre les chaînes de dépendances aux frontières des blocs syntaxiques (limiter la clôture transitive)

Résultat : sur le type de programmes considérés

- nombre d'octogones linéaire en $|\mathbb{V}|$ et donc $|P|$ (taille du programme)
- taille des octogones constante, petite ($\simeq$4)

Exemple : arbres de décision booléens

Le flot de contrôle est souvent encodé dans une variable booléenne

Exemple

$B = (X > 0)$;

⋮ *long code*

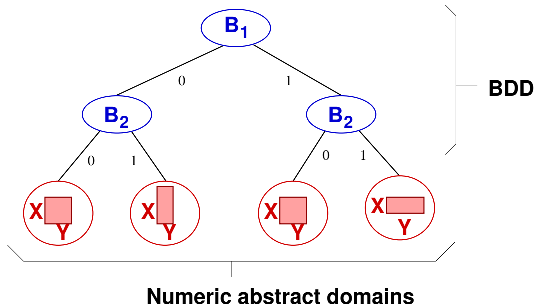
⋮

`if` (B) $Y = 1/X$;

$(B = 1 \wedge X > 0) \vee (B = 0 \wedge X \leq 0)$ non convexe

Il est nécessaire de partitionner X par rapport à la valeur de B

Exemple : arbres de décision booléens



- variables booléennes aux nœuds
- domaines numériques aux feuilles (intervalles, octogones)
- partage des sous-arbres identiques ⇒ meilleur efficacité

Mais il y a trop de variables booléennes pour un seul arbre :

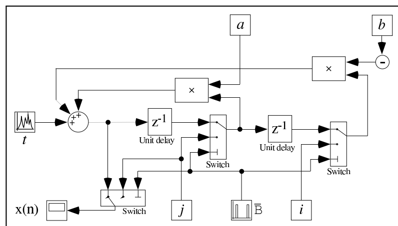
- utiliser un grand nombre d'arbres de petite taille (profondeur 3)
- utiliser un critère syntaxique pour choisir les variables à relier

Exemple : filtrage numérique

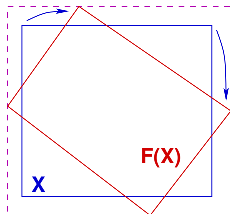
filter.c

```
int INIT = 1;
float P, X, E1,E2, S1,S2;

void filtre2 () {
  if (INIT) {
    P = S1 = E1 = X;
  }
  else {
    P = (0.4677826 * X) -
        (E1 * 0.7700725) + (E2 * 0.4344376) +
        (S1 * 1.5419) - (S2 * 0.6740477);
  }
  E2 = E1;
  E1 = X;
  S2 = S1;
  S1 = P;
}

void main () {
  while (1) {
    X = input(); /* [-10,10] */
    filtre2();
    INIT = input(); /* [0,1] */
    wait_tick();
  }
}
```

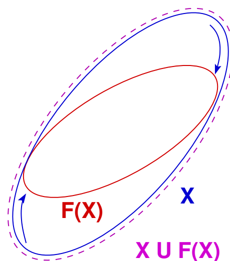
Exemple : filtrage numérique



- calcul de $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- aucun intervalle ou polyèdre n'est stable
- la surface stable la plus simple est une ellipse $Y^2 - aYX - bX^2 \leq c$



(intervalle instable)



(ellipse instable)

Exemple : calculs flottants

Nombres flottants :

- cause d'erreurs additionnelles : dépassements de capacité
- arrondi : sûreté sur les réels $\not\Longrightarrow$ sûreté sur les flottants

  e.g. : `if (x != 0) y = 1 / (x*x)`

Exemple : calculs flottants

Nombres flottants :

- cause d'erreurs additionnelles : dépassements de capacité
- arrondi : sûreté sur les réels $\not\Longrightarrow$ sûreté sur les flottants

  e.g. : `if (x != 0) y = 1 / (x*x)`

Rendre les domaines sûrs sur les flottants :

- Intervalles
  - facile d'être sûr : arrondi vers l'extérieur

    $[a, b] \oplus [c, d] \stackrel{\text{def}}{=} [a \oplus_{-\infty} c, b \oplus_{+\infty} d]$

Exemple : calculs flottants

Nombres flottants :

- cause d'erreurs additionnelles : dépassements de capacité
- arrondi : sûreté sur les réels $\not\Longrightarrow$ sûreté sur les flottants

  e.g. : `if (x != 0) y = 1 / (x*x)`

Rendre les domaines sûrs sur les flottants :

- Intervalles
  - facile d'être sûr : arrondi vers l'extérieur

    $[a, b] \oplus [c, d] \stackrel{\text{def}}{=} [a \oplus_{-\infty} c, b \oplus_{+\infty} d]$

- Domaines relationnels (octogones, polyèdres)
  - donner une sémantique dans les réels par transformation d'expression en ajoutant explicitement l'effet de l'arrondi :

    $X \oplus Y \longrightarrow [1-\epsilon, 1+\epsilon]X + [1-\epsilon, 1+\epsilon]Y + [-\varepsilon, \varepsilon]$ (abstraction)
  - passer ces expressions réelles à un domaine sûr pour les réels
  - le domaine lui-même peut être implanté de manière sûr en flottants

    $X + Y \leq c \wedge -Y \leq d \longrightarrow X \leq c \oplus_{+\infty} d$ (arrondi vers l'extérieur)

Exemple : exponentielles

round.c

```
void main() {
  float X = input(); /* [0,100] */
  while (1) {
    X = X / 101.;
    ...
    X = X * 101.;
    wait_tick();
  }
}
```

Problème :

En flottants $(X/101) \times 101 \neq X$.
$\times$ et $/$ ajoutent une erreur d'arrondi ; X peut croître un peu à chaque itération
$\Longrightarrow$ la borne de X croît exponentiellement !

En pratique, la croissance est suffisamment lente et la durée d'exécution petite pour assurer qu'il n'y a pas de dépassement de capacité

Solution :

Domaine relationnel non linéaire pour relier X et `tick`

$$|X| \leq \alpha(1+a)^{\text{tick}} + \beta \qquad (\alpha,\ \beta \text{ et } a \text{ sont automatiquement inférés})$$

Réduction : `tick` $\leq$ `MAX_TICK` $\Rightarrow$ borne sur X

Exemple : calculs entiers modulaires

Compute-through-overflow

```
signed char x, y; /* in [-1,1] */
(signed char) ( (unsigned char) x + (unsigned char) y )
```

Exemple : calculs entiers modulaires

Compute-through-overflow

```
signed char x, y; /* in [-1,1] */
(signed char) ( (unsigned char) x + (unsigned char) y )
```

Sémantique concrète :

- conversion `signed char` → `unsigned char`
  $\Longrightarrow$ débordement modulaire, qui change $\{-1, 0, 1\}$ en $\{0, 1, 255\}$
- promotion entière : `unsigned char` → `int`
  $\Longrightarrow$ préserve la valeur
- addition dans le type `int` : $\Longrightarrow$ $\{0, 1, 2, 255, 256, 510\}$
- conversion `int` → `signed char`
  $\Longrightarrow$ débordement modulaire, qui retourne $\{-2, -1, 0, 1, 2\}$

Exemple : calculs entiers modulaires

Compute-through-overflow

```
signed char x, y; /* in [-1,1] */
(signed char) ( (unsigned char) x + (unsigned char) y )
```

Sémantique dans les intervalles :

- conversion `signed char` $\rightarrow$ `unsigned char`
  $\Longrightarrow$ débordement modulaire, qui change $[-1, 1]$ en $[0, 255]$
  $\Longrightarrow$ sûr, mais toute la précision est perdue !
- le résultat final est $[-128, 127]$ (imprécis)

Problème :

Le résultat final $[-2, 2]$ est représentable dans les intervalles mais pas les calculs intermédiaires !

Exemple : calculs entiers modulaires

Compute-through-overflow

```
signed char x, y; /* in [-1,1] */
(signed char) ( (unsigned char) x + (unsigned char) y )
```

Solution : domaine des intervalles modulaires

invariants de la forme $[\ell, h] + k\mathbb{Z}, k \in \mathbb{N}$ (valeur à modulo près, k est inféré)

- conversion `signed char` $\rightarrow$ `unsigned char`
  $\Longrightarrow$ débordement, transforme $[-1, 1]$ en $[-1, 1] + 256\mathbb{Z}$
- promotion entière : `unsigned char` $\rightarrow$ `int`
  $\Longrightarrow$ préserve la valeur
- addition dans le type `int` : $\Longrightarrow [-2, 2] + 256\mathbb{Z}$
- conversion `int` $\rightarrow$ `signed char`
  $\Longrightarrow$ débordement, retourne $[-2, 2]$

Exemple : modèle mémoire bas niveau

Type union

```
union {
  struct { uint8 al,ah,bl,bh } b;
  struct { uint16 ax,bx } w;
} r;
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

Type-punning

```
uint8 buf[4] = { 1,2,3,4 };
uint32 i = *((uint32*)buf);
```

Copie rapide

```
float a,b;
*((int*)&a) = *((int*)&b);
```

Norme C : programmes mal typés, comportements indéfinis

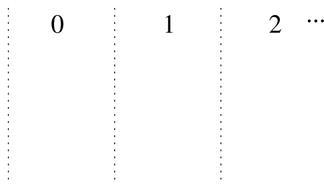
En pratique :

- il n'y a pas d'erreur
- la sémantique est bien définie (spécification de l'*ABI*)

Exemple : modèle mémoire bas niveau

Type union

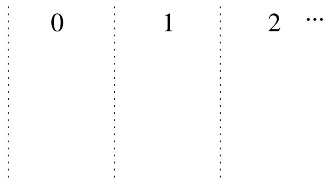
```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```



Principe : sémantique concrète

- découper la mémoire en cellules représentant un entier de 1 octet ou plus
- ajouter (et supprimer) dynamiquement les cellules à l'exécution en fonction des accès par pointeur
- un octet peut être couvert par plusieurs cellules
  $\Longrightarrow$ conjonction de contraintes
  $\Longrightarrow$ possibilité de synthétiser la valeur d'une nouvelle cellule à partir des cellules déjà présentes
- tout octet n'est pas forcément couvert ($\top$)

Exemple : modèle mémoire bas niveau



Type union

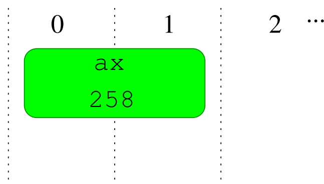
```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

état initial : pas de cellule ($\top$)

Exemple : modèle mémoire bas niveau

Type union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

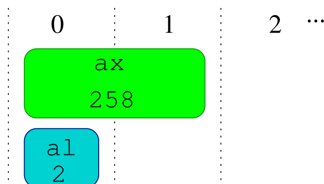


créer `r.w.ax` : cellule `uint16` à l'offset 0

Exemple : modèle mémoire bas niveau

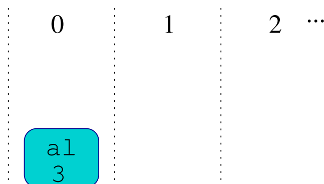
Type union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```



créer `r.b.al` : cellule `uint8` à l'offset 0
initialisée avec : `r.w.ax` mod 256

Exemple : modèle mémoire bas niveau



Type union

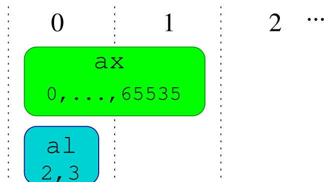
```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

modifier la cellule `r.b.al`
détruire la cellule invalide `r.w.ax`

Exemple : modèle mémoire bas niveau

Type union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

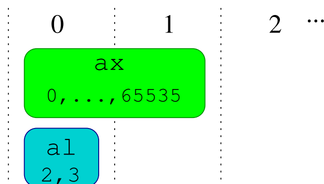


matérialiser les cellules de toutes les branches
et faire l'union des environnements possibles

Exemple : modèle mémoire bas niveau

Type union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```



Dans l'abstrait : associer à chaque cellule

- un intervalle qui sur-approxime l'ensemble de ses valeurs possibles
- ou, plus généralement, une dimension dans un domaine numérique
  e.g. polyèdres, pour garder des relations entre cellules

Exemple : manipulation de flottants au niveau du bit

Conversion

```
double cast(int i) {
  union { int i[2]; double d; } x, y;
  x.i[0] = 0x43300000; y.i[0] = x.i[0];
  x.i[1] = 0x80000000; y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

Exemple : manipulation de flottants au niveau du bit

Conversion

```
double cast(int i) {
  union { int i[2]; double d; } x, y;
  x.i[0] = 0x43300000; y.i[0] = x.i[0];
  x.i[1] = 0x80000000; y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

- `0x43300000 0x80000000` représente $2^{52} + 2^{31}$
- `0x43300000 0x80000000 ^i` représente $2^{52} + 2^{31} + i$
- `y.d - x.d` est égal à `i`
  $\Longrightarrow$ conversion d'entier 32-bit signé vers flottant 64-bit

Justification :

- certains CPU n'ont pas nativement cette instruction (PowerPC)
- on ne veut pas faire confiance au compilateur pour l'émuler
  (tracabilité du code)

Exemple : manipulation de flottants au niveau du bit

Conversion

```
double cast(int i) {
  union { int i[2]; double d; } x, y;
  x.i[0] = 0x43300000; y.i[0] = x.i[0];
  x.i[1] = 0x80000000; y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

Principe d'analyse :

- le domaine mémoire détecte l'utilisation d'une union
  initialisation intelligente à la matérialisation de la cellule
  $\mathtt{y.d} = \textit{dbl_of_word}(\mathtt{y.i[0]}, \mathtt{y.i[1]})$
- un domaine symbolique ad-hoc maintient des prédicats
  - $\mathtt{V} = \mathtt{W\hat{}0x80000000}$ $(\mathtt{y.i[1] = i\hat{}x.i[1]})$
  - $\mathtt{V} = \textit{dbl_of_word}(\mathtt{0x43300000}, \mathtt{W})$ $(\mathtt{y.d})$

  en suivant les affectations, tests, unions, etc.
  (version symbolique de la propagation des constantes : *variable* = *expression*)

Exemple : manipulation de flottants au niveau du bit

Conversion

```
double cast(int i) {
  union { int i[2]; double d; } x, y;
  x.i[0] = 0x43300000; y.i[0] = x.i[0];
  x.i[1] = 0x80000000; y.i[1] = i ^ x.i[1];
  return y.d - x.d;
}
```

réductions entre intervalles et prédicats :

- les prédicats sont inférés par *pattern-matching* des expressions et grâce aux valeurs trouvées dans les intervalles

  (0x43300000, 0x80000000)

- application de règles de réecriture pour affiner les intervalles

  $(\texttt{y.d} - \texttt{x.d} \rightsquigarrow (\texttt{double})\texttt{i})$

facile à étendre à d'autres prédicats et d'autres règles de propagation

Applications d'Astrée

Airbus A340-300 (2003)

Airbus A380 (2004)

(modèle de) ESA ATV (2008)

- taille : de 70 000 à 860 000 lignes de C
- temps d'analyse : de 45mn à $\simeq$40h
- 0 alarme : preuve d'absence d'erreur à l'exécution

L'analyseur statique AstréeA

Les logiciels concurrents

Programmation concurrente :

décomposer un programme en un ensemble de processus qui interagissent

- exploitation du parallélisme des ordinateurs (multi-cœurs, cloud)
- décomposition du programme en tâches asynchrones (serveurs, GUI, programmes réactifs, . . .)

Dans l'avionique : *Integrated Modular Avionics*

- intégrer des fonctionnalités (moins de CPUs)
- remplacer les bus physiques par une mémoire partagée (moins de fils)
- pour les tâches moins critiques (DAL C–E, certification plus légère)
- allocation statique des ressources (threads, locks, mémoire)
- ordonnancement temps-réel (ARINC 653, POSIX threads real-time)

Difficultés :

- les logiciels concurrents sont plus difficiles à concevoir correctement
- et plus difficiles à valider et vérifier
- le test est inefficace,
  les méthodes formelles sûres, sur le source sont peu développées

Sémantique informelle des programmes *multi-thread*

t_1	t_2
$^{\ell 1a}$**while** random **do**	$^{\ell 1b}$**while** random **do**
$^{\ell 2a}$**if** $x < y$ **then**	$^{\ell 2b}$**if** $y < 100$ **then**
$^{\ell 3a}x \leftarrow x + 1$	$^{\ell 3b}y \leftarrow y + [1, 3]$

Modèle d'exécution :

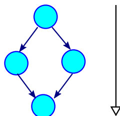
- ensemble fini de *threads* (logiciels embarqués)
- la mémoire est partagée (x,y)
- chaque thread a son propre compteur de programme
- l'exécution entrelace des pas de chaque thread t_1 and t_2
  en choisissant arbitrairement la prochaine thread à exécuter
  (les affectations et les tests sont considérés comme atomiques)

$\Longrightarrow$ nous avons l'invariant global suivant : $0 \leq x \leq y \leq 102$

Retour sur l'analyse séquentielle

Deux méthodes d'analyse :

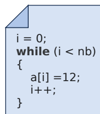
Équationnelle :



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

- dériver un système d'équations du CFG
  une variable $X_i \in \mathcal{D}^\sharp$ par point de programme i
- itérer (propager) le long du CFG
- mémoire linéaire en la taille du programme
- stratégie de propagation flexible
- facile à adapter aux programmes concurrents
  avec un produit de CFG

Dénotationnelle :



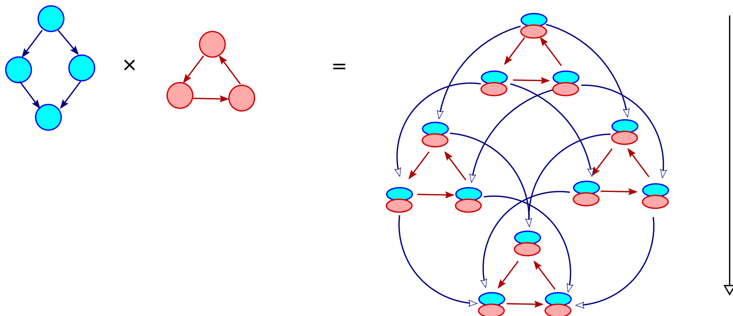
$$\mathsf{C}[\![\, \texttt{while}\ c\ \texttt{do}\ b\ \texttt{done} \,]\!]\, X \stackrel{\text{def}}{=} \mathsf{C}[\![\, \neg c? \,]\!]\, (\mathrm{lfp}\ \lambda Y.X \cup \mathsf{C}[\![\, b? \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, Y))$$

$$\mathsf{C}[\![\, \texttt{if}\ c\ \texttt{then}\ t\ \texttt{fi} \,]\!]\, X \stackrel{\text{def}}{=} \mathsf{C}[\![\, t \,]\!]\, (\mathsf{C}[\![\, c? \,]\!]\, X) \cup \mathsf{C}[\![\, \neg c? \,]\!]\, X$$

...

- méthode vue en cours et en TME
- itération sur la structure syntaxique
- mémoire linéaire en la profondeur du programme
- stratégie d'itération fixée
  (suit la structure du programme)
- pas de définition inductive du produit...

Produit de CFG



Produit des CFG des threads du programme :

- état de contrôle = tuple de points de programmes
  $\Longrightarrow$ explosion combinatoire des états abstraits
- duplication des fonctions de transfert

$\Longrightarrow$ exponentiel en le nombre de threads

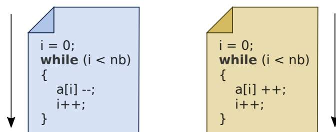
Produit de CFG (exemple)

t_1	t_2
$^{\ell 1a}$**while** random **do**	$^{\ell 1b}$**while** random **do**
$^{\ell 2a}$**if** $x < y$ **then**	$^{\ell 2b}$**if** $y < 100$ **then**
$^{\ell 3a}x \leftarrow x + 1$	$^{\ell 3b}y \leftarrow y + [1, 3]$

Système d'équations :

$$
\begin{aligned}
X_{1a,1b} &= I \\
X_{2a,1b} &= X_{1a,1b} \cup \mathrm{C}[\![x \geq y]\!] X_{2a,1b} \cup \mathrm{C}[\![\mathtt{x} \leftarrow \mathtt{x} + 1]\!] X_{3a,1b} \\
X_{3a,1b} &= \mathrm{C}[\![x < y]\!] X_{2a,1b} \\
X_{1a,2b} &= X_{1a,1b} \cup \mathrm{C}[\![y \geq 100]\!] X_{1a,2b} \cup \mathrm{C}[\![\mathtt{y} \leftarrow \mathtt{y} + [1,3]]\!] X_{1a,3b} \\
X_{2a,2b} &= X_{1a,2b} \cup \mathrm{C}[\![x \geq y]\!] X_{2a,2b} \cup \mathrm{C}[\![\mathtt{x} \leftarrow \mathtt{x} + 1]\!] X_{3a,2b} \cup \\
&\quad X_{2a,1b} \cup \mathrm{C}[\![y \geq 100]\!] X_{2a,2b} \cup \mathrm{C}[\![\mathtt{y} \leftarrow \mathtt{y} + [1,3]]\!] X_{2a,3b} \\
X_{3a,2b} &= \mathrm{C}[\![x < y]\!] X_{2a,2b} \cup X_{3a,1b} \cup \mathrm{C}[\![y \geq 100]\!] X_{3a,2b} \cup \mathrm{C}[\![\mathtt{y} \leftarrow \mathtt{y} + [1,3]]\!] X_{3a,3b} \\
X_{1a,3b} &= \mathrm{C}[\![y < 100]\!] X_{1a,2b} \\
X_{2a,3b} &= X_{1a,3b} \cup \mathrm{C}[\![x \geq y]\!] X_{2a,3b} \cup \mathrm{C}[\![\mathtt{x} \leftarrow \mathtt{x} + 1]\!] X_{3a,3b} \cup \mathrm{C}[\![y < 100]\!] X_{2a,2b} \\
X_{3a,3b} &= \mathrm{C}[\![x < y]\!] X_{2a,3b} \cup \mathrm{C}[\![y < 100]\!] X_{3a,2b}
\end{aligned}
$$

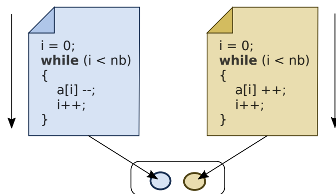
Analyse thread à thread avec interférences simples



Principe : éviter l'explosion combinatoire du contrôle

- analyser chaque thread séparément

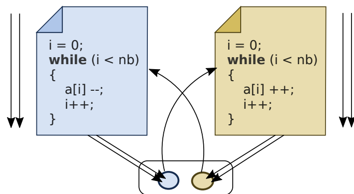
Analyse thread à thread avec interférences simples



Principe : éviter l'explosion combinatoire du contrôle

- analyser chaque thread séparément
- collecter les valeurs écrites dans chaque variable par chaque thread
  $\Longrightarrow$ les interférences
  abstraites dans un domaine abstrait, e.g., les intervalles

Analyse thread à thread avec interférences simples



Principe : éviter l'explosion combinatoire du contrôle

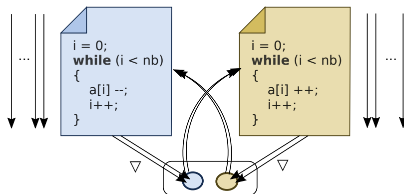
- analyser chaque thread séparément
- collecter les valeurs écrites dans chaque variable par chaque thread $\Longrightarrow$ les interférences

  abstraites dans un domaine abstrait, e.g., les intervalles
- réanalyser les threads, en injectant ces valeurs à chaque lecture

  lire une variable retourne la dernière valeur lue,
  ou une interférence, de manière non-déterministe

  $\Longrightarrow$ de nouveaux états et interférences sont découverts

Analyse thread à thread avec interférences simples



Principe : éviter l'explosion combinatoire du contrôle

- analyser chaque thread séparément
- collecter les valeurs écrites dans chaque variable par chaque thread
  $\Longrightarrow$ les interférences
  abstraites dans un domaine abstrait, e.g., les intervalles
- réanalyser les threads, en injectant ces valeurs à chaque lecture
  lire une variable retourne la dernière valeur lue,
  ou une interférence, de manière non-déterministe
  $\Longrightarrow$ de nouveaux états et interférences sont découverts
- itérer les analyses jusqu'à la stabilisation
  en appliquant un élargissement $\triangledown$ sur les interférences

Exemple d'analyse thread à thread

t_1

$^{\ell 1a}$**while** random **do**
$^{\ell 2a}$**if** $x < y$ **then**
$^{\ell 3a}x \leftarrow x + 1$

t_2

$^{\ell 1b}$**while** random **do**
$^{\ell 2b}$**if** $y < 100$ **then**
$^{\ell 3b}y \leftarrow y + [1, 3]$

Analyse concrète de t_1

(1a): $x = y = 0$
(2a): $x = y = 0$
(3a): $\bot$

Exemple d'analyse thread à thread

t_1

$^{\ell 1a}$**while** random **do**
  $^{\ell 2a}$**if** $x < y$ **then**
    $^{\ell 3a}$$x \leftarrow x + 1$

t_2

$^{\ell 1b}$**while** random **do**
  $^{\ell 2b}$**if** $y < 100$ **then**
    $^{\ell 3b}$$y \leftarrow y + [1, 3]$

Analyse concrète de t_2

$(1b)$: $x = y = 0$
$(2b)$: $x = 0,\ y \in [0, 102]$
$(3b)$: $x = 0,\ y \in [0, 99]$

interférences découvertes : $y \leftarrow [1, 102]$

Exemple d'analyse thread à thread

t_1

$^{\ell 1a}$**while** random **do**
  $^{\ell 2a}$**if** $x < y$ **then**
    $^{\ell 3a}x \leftarrow x + 1$

t_2

$^{\ell 1b}$**while** random **do**
  $^{\ell 2b}$**if** $y < 100$ **then**
    $^{\ell 3b}y \leftarrow y + [1, 3]$

Nouvelle analyse de t_1 avec interférences causées par t_2

interférences à appliquer : $y \leftarrow [1, 102]$

(1a): $x = y = 0$
(2a): $x \in [0, 102],\ y = 0$
(3a): $x \in [0, 102],\ y = 0$

remplacer $x < y$ par $x < \max(y, [1, 102])$
remplacer $x \geq y$ par $x \geq \min(y, [1, 102])$

interférences découvertes : $x \leftarrow [1, 102]$

les analyses suivantes sont identiques : le point fixe est atteint

Exemple d'analyse thread à thread

t_1

$^{\ell 1a}$**while** random **do**
  $^{\ell 2a}$**if** $x < y$ **then**
    $^{\ell 3a}x \leftarrow x + 1$

t_2

$^{\ell 1b}$**while** random **do**
  $^{\ell 2b}$**if** $y < 100$ **then**
    $^{\ell 3b}y \leftarrow y + [1, 3]$

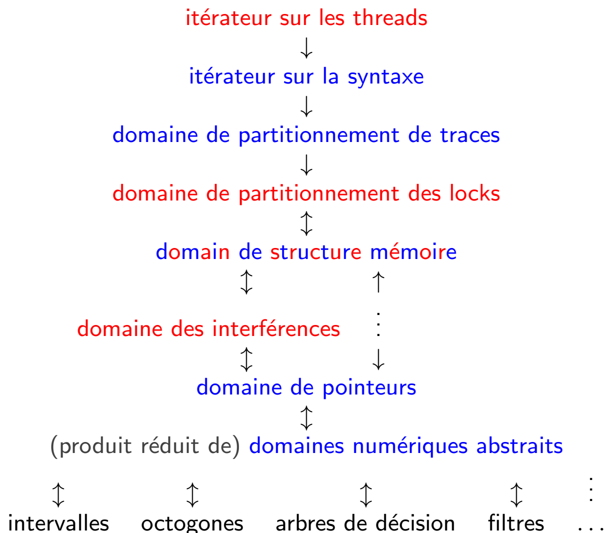
Invariants concrets :

- nous avons $x, y \in [0, 102]$, mais pas $x \leq y$

Analyse abstraite dérivée :

- similaire à une analyse de programmes séquentiels, mais itérée
  paramétrée par un domaine abstrait arbitraire
- efficace (peu de réanalyses nécessaires en pratique)
- les interférences sont non-relationnelles et insensibles au flot de contrôle
  limite héritée de la sémantique concrète

Interprète abstrait d'AstréeA



Applications

Famille cible d'applications

Modèle de concurrence :

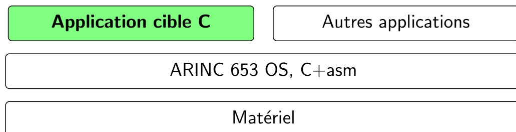
- ensemble fixé de threads
- ordonnancement temps-réel préemptif sur un seul processeur
- mémoire partagée, locks

Application cible :

- code avionique embarqué
- 1.6 Mloc de C, 15 threads
- code réactif + code réseau + listes, chaînes, pointeurs
- nombreuses variables, tableaux, boucles, graphe d'appel peu profond
- pas d'allocation dynamique de mémoire, pas de récursivité

Contexte d'analyse

Environnement d'exécution concrète :

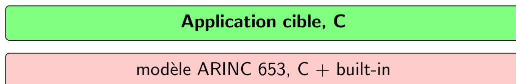


L'application cible :

- s'exécute en concurrence avec d'autres applications (séparation de mémoire)
- interagit dynamiquement avec un OS de type ARINC 653
  (contrôle des threads, mutex lock et unlock, communications)
- interagit avec d'autres applications via l'OS
- crée des objets systèmes seulement pendant la phase d'initialisation
  (l'ensemble des objets créé est inféré par l'analyse de la phase d'initialisation)

Contexte d'analyse

Environnement d'analyse abstraite :



L'application cible est enrichie avec un modèle de l'OS écrit à la main

- 2.6 Kloc de C + built-ins Astrée bas niveau
- simule tous les appels systèmes de l'OS
- implante les objets de l'OS avec des objets bas-niveau d'AstréeA
  (e.g., les mutex d'AstréeA sont de simples entiers, ceux d'ARINC 653 ont un nom chaîne)

$\Longrightarrow$ réduction à l'analyse d'un programme "C" autonome sans symbole indéfini

Résultats

Précision : obtenus par spécialisation

- 2010 : $12,257$ fausses alarmes
- 2015 : $1,195$ fausses alarmes

Efficacité :

- sur une machine intel i7 2.90 GHz
- temps de calcul : 24h
- nombre d'itérations : 6 (pas de widening nécessaire)
- 90 GB RAM

Conclusion

Résumé

Objectif atteint

Astrée : il est possible de construire un analyseur statique à la fois :

- sûr vis à vis d'une sémantique réaliste du C
- raisonnablement efficace en temps et en mémoire
- précis sur une classe infinie de programmes
- utilisable dans un cadre de validation des logiciels critiques

Recette

- partir d'un analyseur simple (intervalles)
- tant qu'il reste des fausses alarmes
  - chercher leur cause et, au choix
    - régler les paramètre d'analyse, ou
    - améliorer un domaine existant, ou
    - ajouter une réduction entre domaines existants, ou
    - ajouter un nouveau domaine