Soutenance de thèse de doctorat présentée à l'Université Pierre et Marie Curie

Langages Applicatifs et Machines Abstraites
pour la Couverture de Code Structurelle

Philippe WANG
sous la direction d'Emmanuel CHAILLOUX

Université Pierre et Marie Curie
École Doctorale Informatique, Télécommunications et Électronique de Paris

Jeudi 4 octobre 2012


APR

CNRS


EDITE

LIP6

UPMC

Contexte : logiciels...

Un logiciel, c'est une suite finie de `0` et de `1` dans un ordre spécifique telle qu'au moins un ordinateur peut l'interpréter et au moins un humain peut l'utiliser.

`01010010001001011010010100101010100101011...`

Faire un logiciel,
c'est fabriquer une bonne séquence de `0` et de `1`.

Se tromper d'un `0` ou d'un `1`,
c'est potentiellement faire n'importe quoi.

e.g., un seul bit de différence : -1066 au lieu de -42 sur une opération bancaire.

« *Un ordinateur, c'est une machine hyper stupide et hyper obéissante.*
C'est le contraire d'un Homme.

Ça va très très vite, ça ne se trompe jamais, c'est stupide. » – Gérard Berry

Contexte : logiciels critiques

Logiciels dont un défaut pourrait causer des décès, des blessures graves, ou des erreurs financières importantes.

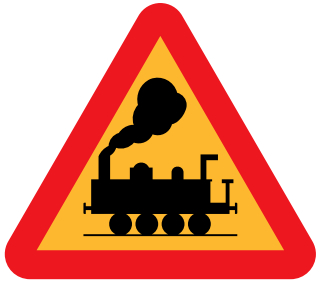
Quelques exemples

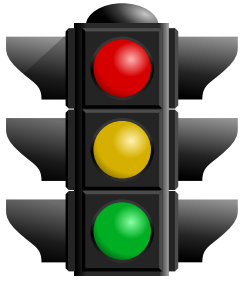
- Transports (*i.e.*, **avionique**, ferroviaire),
- Nucléaire, Médical, Paiement électronique.

Cinq niveaux de criticité

• A/SIL4 • B/SIL3 • C/SIL2 • D/SIL1 • E/SIL0

Très critique (décès) ... Peu critique ... Non critique (Sans effet)





« Faire du code qui marche pas, c'est pas cher.
Faire du code qui marche, c'est pas le même prix.
Il faut choisir ses ennuis. »
– Gérard Berry

SIL : Safety Integrity Level

Langages de programmation

Fabriquer une longue séquence (millions à milliards)

de `0` et de `1` sans se tromper, c'est impossible pour les humains.

Les langages de programmation

servent à exprimer des programmes d'une manière plus abstraite de la machine mais plus proche des langues naturelles.

Les programmes sont traduits (par des compilateurs)

pour devenir des séquences de `0` et de `1`, ce qui donne des binaires exécutables par la machine.

Certes le risque d'erreurs n'est jamais zéro,

toutefois il se réduit quand on utilise des langages bien adaptés à ce qu'on veut faire.

Processus de développement encadrés : « normes »

DO-178B/ED-12B, EN 50128, EN 50128, IEC-61508

avionique civile *ferroviaire* *signalisation* *industrie*

Constat : garantir l'absence de défauts est impossible.
Pour s'assurer de la meilleure qualité pour un logiciel :

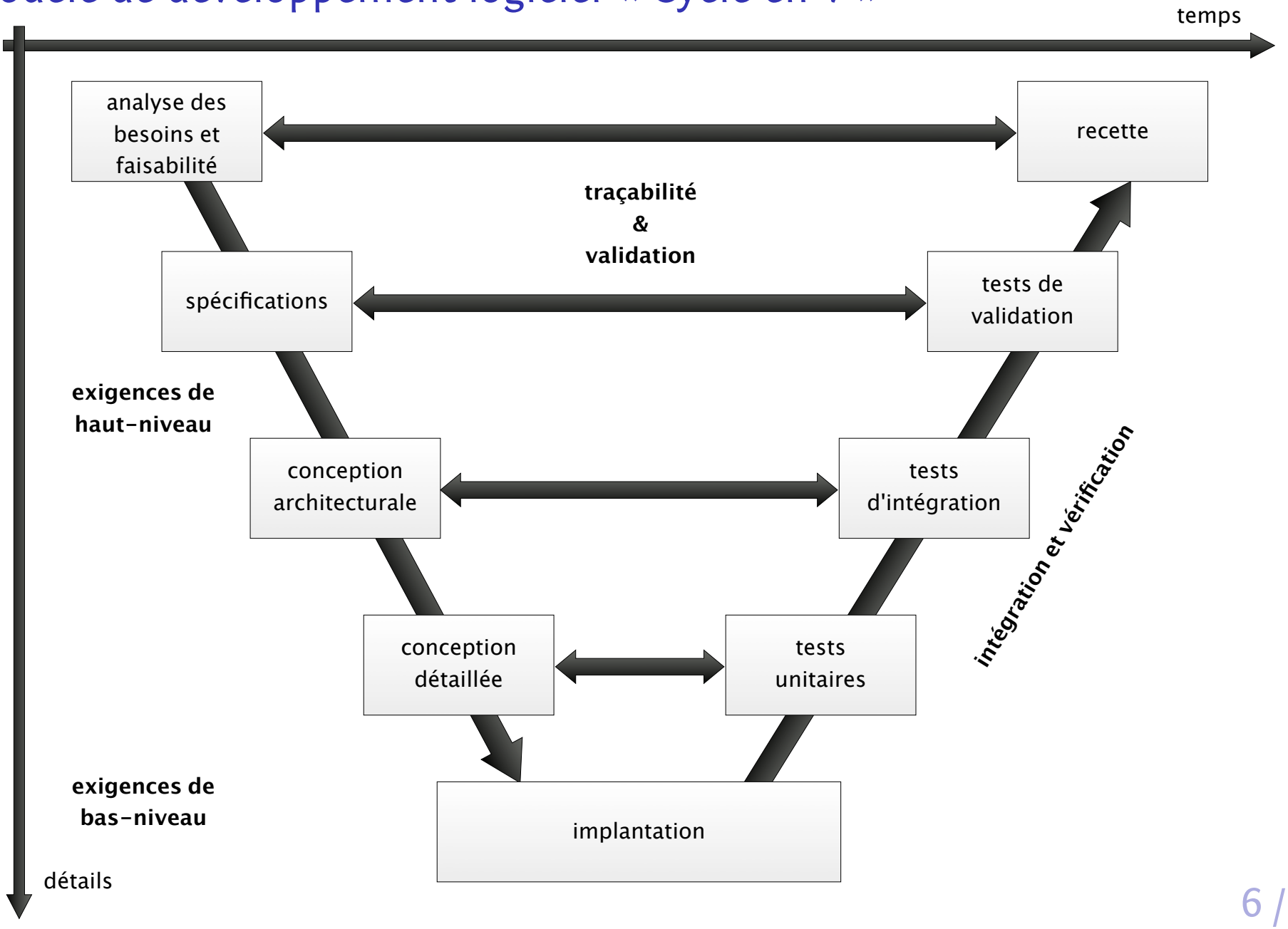
- Processus de génie logiciel bien défini.
- Tout doit toujours être vérifié.
- Respect des objectifs assuré par une autorité indépendante.
- Les normes doivent être acceptées par tous.
- Les entreprises sont responsables des moyens.

Certification de la conformité du processus de production par une **autorité indépendante qui s'engage**.

Processus analogue : Certification AB.

Traçabilité et activités de test, et coûts

Modèle de développement logiciel « Cycle en V »

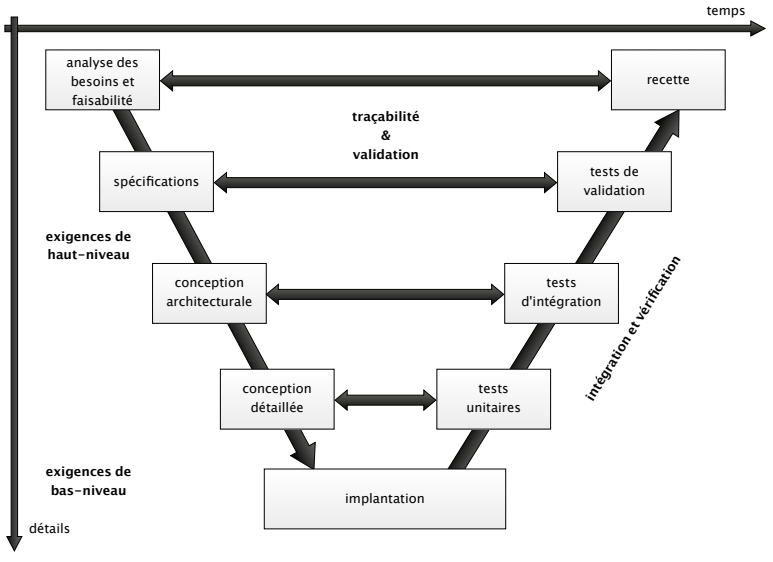


Consolider les étapes de développement

Les spécifications sont la première faille immuable

Chaque étape vers l'implantation est une potentielle source d'introduction d'erreurs humaines

- Minimiser l'intervention humaine dans le raffinement : *automatiser autant que possible*
- Proposer des langages proches de ceux qui expriment les exigences de haut-niveau : *langages de haut-niveau d'abstraction*
- Utiliser des traducteurs automatiques de langages : *compilateurs* ou *générateurs de code*



Exemple : SCADE Suite et KCG

SCADE Suite

- Environnement de développement graphique pour applications critiques.
- Utilisé pour développer des applications de criticité de niveau A.

Générateur de code de SCADE Suite : KCG

- Produit du code de criticité de niveau A.

$\Rightarrow$ Il faut soit **certifier** le code produit par KCG comme s'il avait été écrit par un humain, soit **qualifier** l'outil KCG.

Générateur de code pour des applications critiques
(Outil de développement)

Générateur de code, parcours d'arbres et fonctions récursives : volonté d'utiliser un langage applicatif

outils de développement : droit d'utiliser des allocations mémoires dynamiques.

Pas vraiment dans les habitudes

mais les normes n'imposent ni ne conseillent aucun langage : le choix est laissé aux développeurs.

Appliquer le processus sur le langage choisi

Pour SCADE Suite 6, générateur de code KCG en OCaml : appliquer le processus sur OCaml.

Pourquoi OCaml ?

Langage applicatif de la famille ML

- gestion automatique de la mémoire ;
- adopté pour le développement de KCG ;

———————————————

- typage statique : sûreté d'exécution ;
- types algébriques somme et produit ;
- itérateurs et récursivité : pratiques pour parcourir des arbres ;
- traits impératifs ;
- filtrage par motifs : contrôle de haut-niveau sur les structures de données ;
- exceptions : structure de contrôle.

$\rightsquigarrow$ MCAML, spécifié formellement dans la thèse.

Qualifier un outil écrit en OCaml

Convaincre les autorités de certification

La bibliothèque d'exécution du langage doit satisfaire les exigences de la procédure de qualification

Une bibliothèque d'exécution assez simple pour qu'elle soit qualifiable.

- Gestionnaire mémoire simplifié pour être explicable.
- Traçabilité entre le langage source et le code machine produit par le compilateur.

Besoin d'outils de mesure de couverture pour ce langage

- Développer ces outils pour OCaml
  (objet d'étude principal de la thèse).

Quand OCaml rencontre la couverture de code

- La couverture de code.
- Critères de couverture de code structurelle.
- La couverture de code structurelle pour MCAML.

La couverture de code

Test

- Critère de réussite du test : révéler des défauts.
- Activité permettant d'établir les rapports de couverture.

Couverture de code

- Couverture fonctionnelle : montrer les fonctions testées et leur conformité avec la documentation.
- Couverture structurelle : montrer quelles parties du code ont été activées.
  - Couverture des instructions.
  - Couverture des conditions et des décisions.

$\Rightarrow$ Montrer qu'on a fait tout notre possible pour s'assurer qu'on n'aura pas de mauvaise surprise en production (*e.g.*, code mort qui s'active, décision mal formulée).

Couverture des instructions
(statement coverage)

Montrer que chacune des instructions d'un code a été activée.
À défaut, montrer lesquelles ne l'ont pas été.

```
/* fact(n) returns n! */
int fact(n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return n * fact(n-1);
}
```

```
int s = fact (42);
```

```
/* safe_invert(x) safely
   returns 1/x */
int safe_invert (int x) {
  int r;
  if (x != 0)
    r = 1 / x;
  return r;
}
```

```
int i = safe_invert(2);
```

Conditions et décisions

Selon DO-178B

- **Condition** : une expression booléenne sans opérateur booléen.
- **Décision** : une expression booléenne avec zéro, un ou plusieurs opérateurs booléens.

```
/* fact(n) returns n! */
int fact (int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return n * fact(n-1);
}
```

```
/* safe_invert(x) safely
returns 1/x */
int safe_invert (int x) {
  int r;
  if (x != 0)
    r = 1 / x;
  return r;
}
```

Critères de couverture des conditions et des décisions

Définitions

- **Couverture des conditions (CC)**
  Chacune des conditions a pris à la fois la valeur vraie et la valeur fausse.
- **Couverture des décisions (DC)**
  Chacune des décisions a pris à la fois la valeur vraie et la valeur fausse.
- **Couverture des conditions et décisions (CC+DC=CDC)**
  Chacune des conditions et des décisions a pris à la fois la valeur vraie et la valeur fausse.
- **Couverture des conditions multiples (MCC)**
  Pour chacune des décisions, les conditions qui la composent ont pris toutes les configurations possibles.

Critères de couverture des conditions et des décisions

Exemples avec $a \wedge (b \vee c) = r$

- Couverture des conditions (CC) : 2

```
                    a b c    r
Configuration 1 : T T T → T
Configuration 2 : F F F → F
```

- Couverture des décisions (DC) : 2

```
Configuration 1 : T T T → T
Configuration 2 : F F F → F
```

- Couverture des conditions/décisions (CC+DC=CDC) : 2

```
Configuration 1 : T T T → T
Configuration 2 : F F F → F
```

- Couverture des conditions multiples (MCC) : 2^N

```
Configuration 1 : T T T → T
Configuration 2 : T T F → T
Configuration 3 : T F T → T
Configuration 4 : F T T → F
Configuration 5 : T F F → F
Configuration 6 : F F T → F
Configuration 7 : F T F → F
Configuration 8 : F F F → F
```

Couverture des conditions/décisions modifiée (MC/DC)

Intérêts, faisabilité et coûts des mesures

- 2 tests pour une décision à N conditions, c'est trop peu.
  En fait, 2 tests ne suffisent pas toujours.
- 2^N tests pour une décision à N conditions, c'est trop.
  En fait, 2^N tests ne sont pas toujours faisables.
- MC/DC est un compromis qui répond à ces problèmes.
  Nombre de tests requis : entre N+1 et 2N pour N conditions.

Intérêt/Rôle

- Montrer qu'en pratique, on a pu observer que chacune des conditions d'une décision a un rôle avéré dans le calcul du résultat de la décision : c'est l'indépendance des conditions qui se trouvent au sein d'une décision.

Couverture des conditions/décisions modifiée (MC/DC)

Exemple avec $a \wedge (b \vee c) = r$

Les configurations possibles

	a	b	c		r
Configuration 1 :	T	T	T	→	T
Configuration 2 :	T	T	F	→	T
Configuration 3 :	T	F	T	→	T
Configuration 4 :	F	T	T	→	F
Configuration 5 :	T	F	F	→	F
Configuration 6 :	F	F	T	→	F
Configuration 7 :	F	T	F	→	F
Configuration 8 :	F	F	F	→	F

→ 5 tests

Configurations montrant les rôles avérés

Pour a : c1 et c4 ; pour b : c2 et c5 ; pour c : c3 et c5.

Couverture des conditions/décisions modifiée (MC/DC)

Exemple avec $a \wedge (b \vee c) = r$

Les configurations possibles

	a	b	c		r
Configuration 1 :	T	T	T	→	T
Configuration 2 :	T	T	F	→	T
Configuration 3 :	T	F	T	→	T
Configuration 4 :	F	T	T	→	F
Configuration 5 :	T	F	F	→	F
Configuration 6 :	F	F	T	→	F
Configuration 7 :	F	T	F	→	F
Configuration 8 :	F	F	F	→	F

→ 4 tests

Nombre minimum de configurations possible : n+1

Configurations montrant les rôles avérés

Pour a : **c3 et c6** ; pour b : c2 et c5 ; pour c : c3 et c5.

Opérateurs séquentiels : faisabilité

Exemple avec $a \wedge (b \vee c)$

Les configurations possibles

	a	b	c		r
Configuration 1 :	T	T	–	→	T
Configuration 2 :	T	T	–	→	T
Configuration 3 :	T	F	T	→	T
Configuration 4 :	F	–	–	→	F
Configuration 5 :	T	F	F	→	F
Configuration 6 :	F	–	–	→	F
Configuration 7 :	F	–	–	→	F
Configuration 8 :	F	–	–	→	F

→ 4 tests

Nombre minimum de configurations possible : n+1

Certaines configurations sont factorisées

c1 = c2 ; c4 = c6 = c7 = c8

Conditions et décisions en C

En C, le repérage est syntaxique : il faut que l'expression comporte un opérateur booléen (*e.g.*, `a && b`) ou qu'elle se situe avant un branchement conditionnel (*e.g.*, `if`, `while`), ou encore qu'elle soit le résultat d'un prédicat (*e.g.*, `a == b`).

⇒ **règles de codage**

Déportation et masquage

```
int x = a && b;
int y = c && d;
if (x && y) ...
```

Implantation :

```
int et(int x, int y) {
  return x && y;
}
```

Utilisation :

```
// x et y: des conditions?!
if (et(x, y)) ...
```

Conditions et décisions pour mCaml

En mCaml, les expressions booléennes sont de type `bool`.

Contribution : proposition de quatre sémantiques formelles pour définir les conditions et les décisions pour mCaml :

1. la plus proche de C, avec règles de codage ;
2. léger relâchement des règles de codage, reste classique ;
3. généralisation des opérateurs booléens aux fonctions ;
4. remplacement des règles de codage par des obligations de mesures.

Conditions et décisions pour mCaml

Sémantique 1

Principes

- Les décisions doivent apparaître avant un branchement conditionnel (`if`, `while`)
  et ne peuvent pas apparaître ailleurs.
- Toutes les expressions booléennes doivent être une condition et/ou une décision.
- Un ensemble bien défini d'opérateurs booléens existe (&& et | |), ils permettent de composer les expressions booléennes entre elles.

Des expressions valides en mCaml sont rejetées.

```
let x : bool = (a < 3) && (b > 4)     (* rejetée *)

let y : bool = f (a < 3) (b > 4)      (* rejetée *)
```

Conditions et décisions pour mCaml

Sémantique 2

Principes

- Les expressions booléennes ne peuvent être composées que par l'ensemble bien défini d'opérateurs booléens (&& et ||).
- + On autorise les décisions à apparaître ailleurs que devant un branchement.

Un peu moins d'expressions valides en mCaml sont rejetées.

```
let x : bool = (a < 3) && (b > 4)   (* acceptée *)

let y : bool = f (a < 3) (b > 4)    (* rejetée *)
```

Conditions et décisions pour mCaml

Sémantique 3 : généralisation des opérateurs booléens aux fonctions

Principes

- On autorise les décisions à apparaître ailleurs que devant un branchement.
- $+$ Les expressions booléennes peuvent être composées par n'importe quelle fonction à retour booléen.

```
let x : bool = (a < 3) && (b > 4)     (* acceptée *)
let y : bool = f (a < 3) (b > 4)    (* acceptée *)
```

Des expressions valides en mCaml sont encore rejetées.

```
let z : bool = (e1 ; e2) && e3 (* rejetée *)
let t : bool = (let x = e1 in e2) && e3 (* rejetée *)
let i : bool = if a > 1 then b else c && d (* rejetée *)
```

Conditions et décisions pour mCaml

Sémantique 4

Principes

- Les expressions booléennes peuvent être composées par n'importe quelle fonction à retour booléen.
- $+$ Les expressions booléennes peuvent être composées par d'autres constructions syntaxiques, comme la conditionnelle par exemple.
- $+$ Aucune construction valide rejetée.

```
(* une décision à 4 conditions *)
let i1 = if a > 1 then b else c && d

(* la même mais compatible avec la sémantique 2 *)
let i2 = (a > 1 && b) || (not (a > 1) && c && d)

let z : bool = (e1 ; e2) && e3 (* acceptée *)
let t : bool = (let x = e1 in e2) && e3 (* acceptée *)
```

Obtenir un rapport de couverture de code

Rapport de couverture de code structurelle

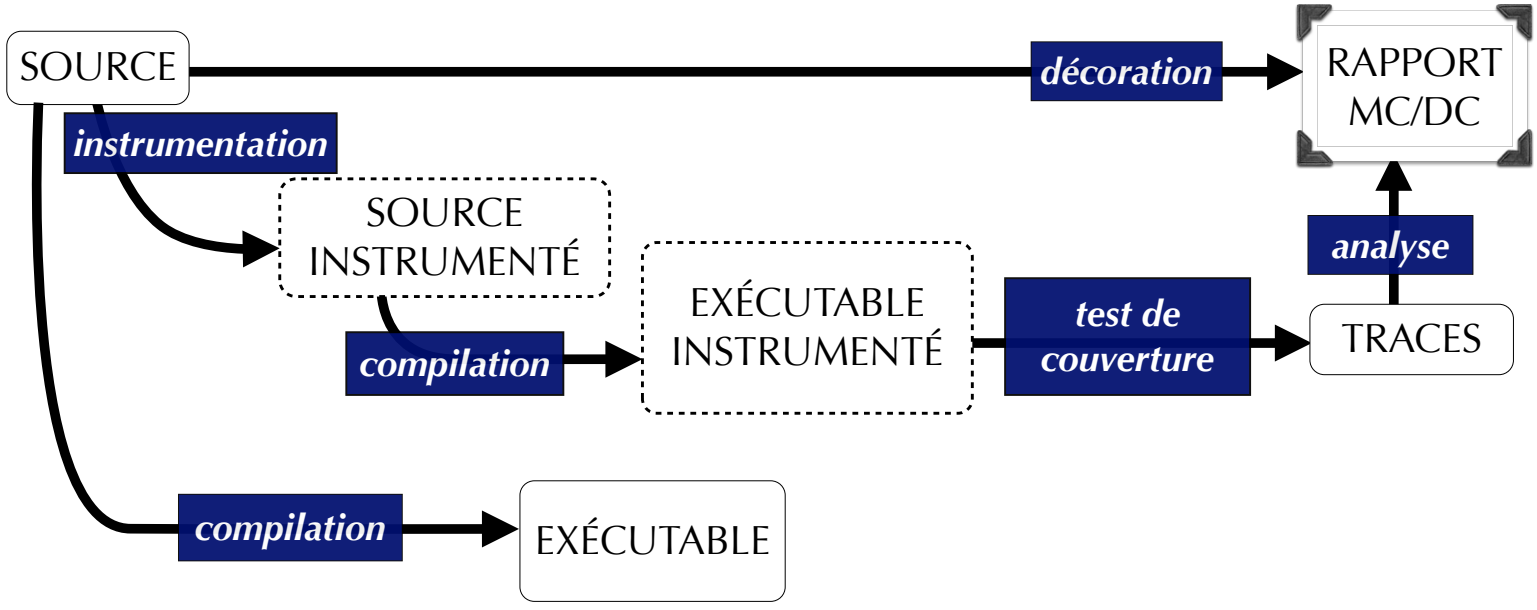
- Montrer les parties couvertes et les parties non couvertes du code.
  - Le code décoré avec les informations.
  - Les mêmes données sous une autre forme.
- Générer des traces d'exécution pour savoir a posteriori ce qui s'est passé (établir la mesure de couverture).

Deux approches pour faire obtenir les traces d'exécution permettant d'établir les rapports

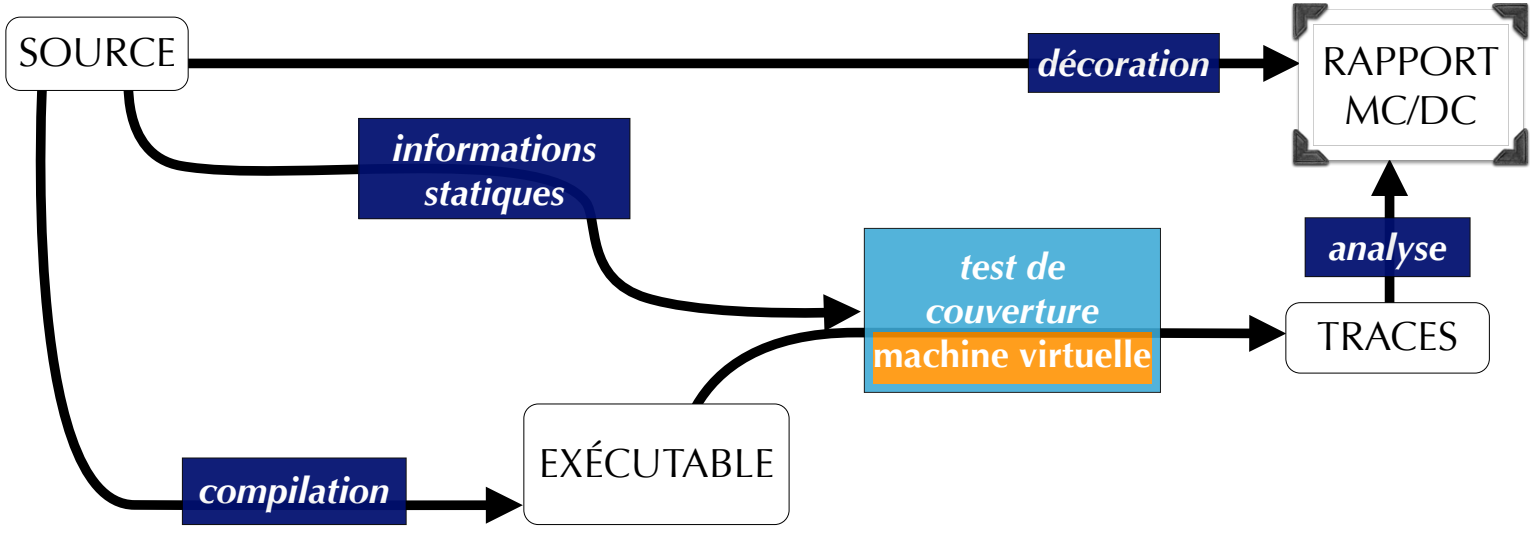
1. Instrumentation intrusive : réécriture du code source.
2. Instrumentation non intrusive : modification de l'environnement d'exécution.

Obtenir un rapport de couverture de code

par l'instrumentation du code source



par l'instrumentation de l'environnement d'exécution



Obtenir les traces d'exécution par l'approche intrusive

Principes

- Réécrire le programme pour lui faire émettre des traces d'exécution comportant les informations voulues.
- Binaire plus volumineux.
- Préservation de la sémantique.

Schéma général de réécriture

$$e \rightsquigarrow \star \texttt{ ; let r = e in } \star \texttt{ ; r}$$

(où $\star$ = une émission de trace)

Traces

```
1,1 – 1,42 : Covered
4,4 – 4,12 : True
...
4,4 – 4,12 : False
```

Approche intrusive

Schéma de réécriture pour la couverture des expressions

Schéma simple

$$
\begin{array}{rcl}
[\![\texttt{atom}]\!] & = & \star\texttt{; atom} \\
[\![\texttt{e1 e2}]\!] & = & \star\texttt{; let r = }[\![\texttt{e1}]\!]\ [\![\texttt{e2}]\!]\texttt{ in }\star\texttt{; r} \\
[\![\texttt{if e1 then e2 else e3}]\!] & = & \texttt{if }[\![\texttt{e1}]\!]\texttt{ then }[\![\texttt{e2}]\!]\texttt{ else }[\![\texttt{e3}]\!] \\
[\![\texttt{fun x -> e}]\!] & = & \texttt{fun x -> }[\![\texttt{e}]\!] \\
[\![\texttt{e1; e2}]\!] & = & [\![\texttt{e1}]\!]\texttt{; }[\![\texttt{e2}]\!] \\
[\![\texttt{e1 \&\& e2}]\!] & = & [\![\texttt{e1}]\!]\texttt{ \&\& }[\![\texttt{e2}]\!] \\
[\![\texttt{match e1 with p -> e2}]\!] & = & \texttt{match }[\![\texttt{e1}]\!]\texttt{ with p -> }[\![\texttt{e2}]\!] \\
[\![\texttt{raise e}]\!] & = & \texttt{raise }[\![\texttt{e}]\!] \\
[\![\texttt{while e1 do e2 done}]\!] & = & \texttt{while }[\![\texttt{e1}]\!]\texttt{ do }[\![\texttt{e2}]\!]\texttt{ done; }\star
\end{array}
$$

Réduire les marqueurs inutiles

Par exemple, pour l'application `(x1 x2)` où `x1` et `x2` sont des variables, nul besoin de marqueurs pour `x1` et `x2`.

Approche intrusive

Schéma de réécriture pour la couverture des conditions et décisions

Hors d'une décision

```
                ᶜ⟦c , ∅⟧   =   c
             ᶜ⟦x : bool , ∅⟧   =   let d = newVector(1) in update(d,x)
          ᶜ⟦e1 e2 : bool, ∅⟧   =   let d = newVector(#(e1 e2))
                                   in update(d, ᶜ⟦e1, d⟧ ᶜ⟦e2 , d⟧)
            ᶜ⟦fun x -> e, ∅⟧   =   fun x -> ᶜ⟦e, ∅⟧
```

Dans une décision

```
                ᶜ⟦c , d⟧   =   c
             ᶜ⟦x : bool , d⟧   =   update(d,x)
          ᶜ⟦e1 e2 : bool, d⟧   =   update(d, ᶜ⟦e1, d⟧ ᶜ⟦e2 , d⟧)
            ᶜ⟦fun x -> e, d⟧   =   fun x -> ᶜ⟦e, ∅⟧
```

Obtenir les traces d'exécution par l'approche non intrusive

Principes

- Générer les informations *i* qui permettent de faire correspondre le code machine avec le code source.
- Modifier l'environnement d'exécution pour produire les traces selon les informations *i*.
- Binaire identique.
- Préservation de la sémantique.
- Utilisation de la machine virtuelle OCaml.

Machine virtuelle OCaml

- Machine à pile, avec quelques registres (ACCU, PC, SP, TRAPSP, EXTRAARGS, ENV, GLOBALDATA)
- Représentation uniforme des données.
- Non typée.
- 147 instructions.
- Mécanisme d'application général (APPLY).
- Instruction spécifique pour les appels terminaux (APPTERM).
- Gestion automatique de la mémoire.

$\rightarrow$ c'est celle qu'on instrumente pour produire les traces équivalentes à l'approche intrusive.

Approche non intrusive

Schéma de génération des informations pour la couverture des expressions

$$
\begin{aligned}
\mathcal{Z}(\texttt{atom}) &= \{\texttt{atom}\} \\
\mathcal{Z}(\texttt{e1 e2}) &= \{\texttt{e1 e2}\} \cup \mathcal{Z}(\texttt{e1}) \cup \mathcal{Z}(\texttt{e2}) \\
\mathcal{Z}(\texttt{if e1 then e2 else e3}) &= \{\texttt{if e1 then e2 else e3}\} \\
&\quad \cup \mathcal{Z}(\texttt{e1}) \cup \mathcal{Z}(\texttt{e2}) \cup \mathcal{Z}(\texttt{e3}) \\
\mathcal{Z}(\texttt{fun x -> e}) &= \mathcal{Z}(\texttt{e}) \\
\mathcal{Z}(\texttt{e1; e2}) &= \mathcal{Z}(\texttt{e1}) \cup \mathcal{Z}(\texttt{e2}) \\
\mathcal{Z}(\texttt{e1 \&\& e2}) &= \{\texttt{e1 \&\& e2}\} \cup \mathcal{Z}(\texttt{e1}) \cup \mathcal{Z}(\texttt{e2}) \\
\mathcal{Z}(\texttt{match e1 with p -> e2}) &= \{\texttt{match e1 with p -> e2}\} \\
&\quad \cup \mathcal{Z}(\texttt{e1})\mathcal{Z}(\texttt{e2}) \\
\mathcal{Z}(\texttt{raise e}) &= \mathcal{Z}(\texttt{e}) \\
\mathcal{Z}(\texttt{while e1 do e2 done}) &= \{\texttt{while e1 do e2 done}\} \\
&\quad \cup \mathcal{Z}(\texttt{e1}) \cup \mathcal{Z}(\texttt{e2})
\end{aligned}
$$

$\{\texttt{e}\}$ = information de liaison entre e et le code machine lui correspondant

Approche non intrusive

Schéma de génération d'informations pour la couverture des conditions et décisions

Hors d'une décision

$$
\begin{aligned}
{}^{c}\mathcal{Z}(\texttt{c , } \emptyset) &= \emptyset \\
{}^{c}\mathcal{Z}(\texttt{x : bool , } \emptyset) &= \{\texttt{newVector(1),x}\} \\
{}^{c}\mathcal{Z}(\texttt{e1 e2 : bool, } \emptyset) &= \{\texttt{d = newVector(\#(e1 e2)),e1 e2}\} \\
&\quad \cup \{\texttt{d, } {}^{c}\mathcal{Z}(\texttt{e1, d})\} \cup \{{}^{c}\mathcal{Z}(\texttt{e2 , d})\} \\
{}^{c}\mathcal{Z}(\texttt{fun x -> e, } \emptyset) &= {}^{c}\mathcal{Z}(\texttt{e, } \emptyset)
\end{aligned}
$$

Dans une décision

$$
\begin{aligned}
{}^{c}\mathcal{Z}(\texttt{c , d}) &= \emptyset \\
{}^{c}\mathcal{Z}(\texttt{x : bool , d}) &= \{\texttt{d,x}\} \\
{}^{c}\mathcal{Z}(\texttt{e1 e2 : bool, d}) &= \{\texttt{d, e1 e2}\} \cup {}^{c}\mathcal{Z}(\texttt{e1, d}) \cup {}^{c}\mathcal{Z}(\texttt{e2 , d}) \\
{}^{c}\mathcal{Z}(\texttt{fun x -> e, d}) &= {}^{c}\mathcal{Z}(\texttt{e, } \emptyset)
\end{aligned}
$$

Préservation de la sémantique

(dépend des schémas de compilation)

Version intrusive

$$\frac{E, M_1, T_1 \vdash \star_1 \rightsquigarrow (), M_1, T_2 \qquad E, M_1, T_2 \vdash e \rightsquigarrow v, M_2, T_2 \qquad E, M_2, T_2 \vdash \star_2 \rightsquigarrow (), M_2, T_3}{E, M_1, T_1 \vdash \star_1 ; \text{ let r = e in } \star_2 ; r \rightsquigarrow v, M_2, T_3}$$

Version non intrusive

$$\frac{E, M_1, T_1, \mathcal{I} \vdash e \rightsquigarrow v, M_2, T_1 \qquad E, M_2, T_1, \mathcal{I} \vdash \mathcal{I}(e, v) \rightsquigarrow v, M_2, T_2}{E, M_1, T_1, \mathcal{I} \vdash e \rightsquigarrow v, M_2, T_2}$$

$\mathcal{I}$ = {newVector(6),e1 e2} ∪ ... ∪ {if x then y else z} ∪ {d,x}...

L'équivalence dépend des optimisations du compilateur

Exemples

L'approche intrusive casse certaines optimisations

- si l'appel `f x` est en position terminale,
  dans `⋆; let r = f x in ⋆; r` il ne l'est pas.

L'approche non intrusive limite les optimisations possibles

- `fact(5)` remplacé par `120` : quel est le taux de couverture de `fact` si `120` est couvert ?
- Si du code machine est partagé, quelle est la partie du source qui est couverte lorsqu'il est exécuté ?

Deux implantations libres : MLcov et Zamcov
pour un sous-ensemble du langage OCaml

MLcov

- développé et distribué par Esterel Technologies ;
- approche intrusive (*patch* du *front-end* OCaml) ;
- utilisé industriellement pour KCG ;
- fonctionne pour une large partie du langage ;
- sémantique conditions/décisions proche de la N°2.

Zamcov, initié par le projet Couverture

- développé en OCaml et distribué par le LIP6 ;
- approche non intrusive (VM OCaml, en OCaml) ;
- prototype testé avec KCG ;
- sémantique conditions/décisions N°4.

Conclusion

mCaml, un important sous-ensemble du langage OCaml spécifié formellement, pour lequel :

- 4 sémantiques formelles pour les notions de « condition » et « décision », s'adapte bien à l'ensemble du langage OCaml ;
- l'approche classique intrusive par réécriture du code source formalisée pour mCaml ;
- l'approche non intrusive sans réécriture du code source formalisée pour mCaml ;
- deux implantations pour OCaml (MLcov et Zamcov) comparées.

Perspectives

de plus en plus d'outils, pour développer des applications de plus en plus complexes...

- Étudier la pertinence de traiter le filtrage par motifs comme des décisions.
- Étudier le *dispatch* (envoi de messages) (DO-178C).
- Polymorphisme paramétrique.
- Sémantique pour mesures sur des programmes concurrents.
- Réduire les obligations de traces : déduire certaines traces à partir d'autres.
- Sémantique des conditions et décisions pour d'autres langages multiparadigmes en essayant de préserver leur expressivité (*e.g.*, Scala).

Merci