

Introduction aux méthodes formelles

ESIR 3 – TQL – 29 novembre 2011

David MENTRÉ – dmentre@linux-france.org

Plan de la présentation

- ▶ Aperçu général
 - ▶ *Pourquoi* utiliser les méthodes formelles et comment ?
- ▶ Analyse abstraite
 - ▶ Démonstration *automatique* de propriétés sur du code réel
- ▶ Model checking
 - ▶ Vérifier des propriétés *temporelles* sur des automates
- ▶ Logique de Hoare
 - ▶ Prouver des propriétés génériques *sur un programme* complet
- ▶ Démonstrateurs interactifs de théorèmes
 - ▶ Toute la *puissance des mathématiques*... et sa complexité
- ▶ Conclusion





Prélude : deux exemples
illustratifs

Ce code contient une erreur !

- ▶ Calcul de la valeur absolue d'un nombre en langage C

```
int z_abs_x(const int x)
{
    int z;
    if (x < 0)
        z = -x;
    else /* x >= 0 */
        z = x;
    return z;
}
```



Ce code contient une erreur !

- ▶ Calcul de la valeur absolue d'un nombre en langage C

```
int z_abs_x(const int x)
{
    int z;
    if (x < 0)
        z = -x;
    else /* x >= 0 */
        z = x;
    return z;
}
```

- ▶ Solution : si $x = -2^{31}$?
 - ▶ 2^{31} n'existe pas, seulement $2^{31}-1$
-



Un autre exemple, en Java

► Recherche par dichotomie dans un tableau trié

```
► public static int binarySearch(int[] a, int key) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        int midVal = a[mid];  
  
        if (midVal < key)  
            low = mid + 1  
        else if (midVal > key)  
            high = mid - 1;  
        else  
            return mid; // key found  
    }  
    return -(low + 1); // key not found.  
}
```



Un autre exemple, en Java

▶ Recherche par dichotomie dans un tableau trié

```
▶ public static int binarySearch(int[] a, int key) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2; // dépassement si low + high > 231-1  
        int midVal = a[mid];  
  
        if (midVal < key)  
            low = mid + 1  
        else if (midVal > key)  
            high = mid - 1;  
        else  
            return mid; // key found  
    }  
    return -(low + 1); // key not found.  
}
```

▶ Solution

```
▶ 6: int mid = low + ((high - low) / 2);
```

▶ Problème

- ▶ Bug présent de le JDK de Sun/Oracle ! Il a impacté des utilisateurs
- ▶ <http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>



Introduction générale

“Program testing can be used to show the presence of bugs, but never show their absence” – Edsger W. Dijkstra

Que sont les méthodes formelles ?

▶ Principe des méthodes formelles

- ▶ Utiliser les **mathématiques** pour concevoir et si possible réaliser des systèmes informatiques

- ▶ **Spécification** formelle / **Vérification** formelle / **Synthèse** formelle

- ▶ Avantage : **non ambiguïté** des mathématiques !

▶ Objectif global : **améliorer la confiance** !

- ▶ Dans le logiciel et le matériel

- ▶ « *Program testing can be used to show the presence of bugs, but never show their absence* » – Edsger W. Dijkstra

- ▶ Pas une « *silver bullet* » !

- ▶ « *Beware of bugs in the above code; I have only proved it correct, not tried it.* » – Donald E. Knuth (dans le code source de TeX)



Approche générale

▶ En plusieurs étapes

1. *Spécifier* le logiciel en utilisant les mathématiques
2. *Vérifier* certaines propriétés sur cette spécification
 - ▶ Corriger la spécification si besoin
3. (parfois) *Raffiner* ou *dériver* de la spécification un logiciel concret
4. (ou parfois) Faire un *lien* entre la spécification et (une partie d') un logiciel concret

▶ De nombreux formalismes (pour le moins !)

- ▶ Spécification algébrique, Machines d'état abstraites, Interprétation abstraite, *Model Checking*, Systèmes de types, Démonstrateurs de théorèmes automatiques ou interactifs, ...
- ▶ Nous en présenterons certains par des *exemples*



Pourquoi utiliser des méthodes formelles ?

- ▶ Re-formuler la spécification en utilisant les mathématiques force à *être précis*
 - ▶ Un moyen d'avoir des *spécifications claires*
 - ▶ On explicite le « *quoi* » mais pas le « *comment* »
 - ▶ Au passage : aussi *crucial* d'expliquer le « *pourquoi* », documentez !
- ▶ Avec des outils *automatiques* ou des vérifications *manuelles*, fournir des *preuves de fiabilité*
 - ▶ 60 à 80% du coût total est la *maintenance* (source Microsoft)
 - ▶ 20 fois plus cher de gérer un bug en production plutôt qu'en conception
 - ▶ Exemples célèbres : *Pentium FDIV bug* (Jan. 1995, pre-tax charge of \$475 million for Intel), *Therac 25* (5 morts), *Ariane 501* (~\$220 million), ...



Peut-on utiliser les méthodes formelles ?

- ▶ La *maturité* des outils et leurs usages s'est beaucoup amélioré ces dernières années
 - ▶ Pas besoin d'avoir une thèse pour les utiliser
 - ▶ Exemple : ingénieurs de ClearSy (Méthode B)
 - ▶ Mais certains sont toujours très complexes
- ▶ La plupart des outils sont *disponibles gratuitement* et/ou *librement*
 - ▶ <http://gulliver.eu.org/wiki/FreeSoftwareForFormalVerification>
 - ▶ <http://www.atelierb.eu/telecharger-latelier-b/>



Comment les appliquer ?

- ▶ Quel est votre *problème* ? Qu'est-ce que vous voulez *garantir* ?
 - ▶ Trouver une méthode formelle qui corresponde
 - ▶ À votre domaine d'application
 - ▶ À vos problèmes
 - ▶ À vos contraintes de temps et de coûts
 - ▶ ...
- ▶ Comment les mettre en œuvre ?
 - ▶ Les *intégrer* à votre *cycle de développement*
 - ▶ Retour à moyen/long terme
 - ▶ Par ex. : mettre à jour la spécification formelle quand la spécification ou le système réalisé changent
 - ▶ Prendre les *conseils d'un expert* dans la méthode choisie



Une grille de lecture des technologies

- ▶ **Domaines** d'application / **Problèmes** possibles
 - ▶ Quand utiliser cette approche ? Points sensibles à regarder
- ▶ Niveau d'**expertise** / Niveau d'**intervention**
 - ▶ Nul (cliquer un bouton) / Moyen (écrire une spec formelle) / Elevé (faire une preuve)
 - ▶ Que faut-il faire (modèle, annotations, propriétés, ...) ?
- ▶ **Couverture** du cycle de développement / **Fidélité** au logiciel
 - ▶ À quelles étapes du cycle de développement ?
 - ▶ Vérifications sur un modèle du logiciel ou le logiciel lui-même ?
- ▶ **Disponibilité** des outils / Niveau d'**automatisme**
- ▶ **Expressivité** : qu'est-ce que je peux prouver ?



Utile le formel ? (1 / 3)

▶ Le formel est vraiment *utile* ? Juste un *exemple*...

▶ <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

Paris, 19 July 1996

ARIANE 5

Flight 501 Failure

Report by the Inquiry Board



Utile le formel ? (2/3)

► Chaîne d'événements techniques

Based on the extensive documentation and data on the Ariane 501 failure made available to the Board, the following chain of events, their inter-relations and causes have been established, starting with the destruction of the launcher and tracing back in time towards the primary cause.



- The launcher started to disintegrate at about $H_0 + 39$ seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher.
- This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine.
- These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.
- The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception.

Exception à l'exécution



Utile le formel ? (3 / 3)

Pourquoi
l'erreur s'est
propagée

► Chaîne d'événements techniques

[...]

Cause
informatique
de l'exception

- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

[...]

Problème de
spécification

- The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time.
- The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values.



Recommandations pour Ariane 501

▶ Plus de *tests* !

R2 Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained.

▶ Plus de *formel* (implicite) !

R5 Review all flight software (including embedded software), and in particular :

- Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment.
- Verify the range of values taken by any internal or communication variables in the software.

▶ ⇒ Utilisation d'*analyse abstraite* (PolySpace, Alain Deutsch)





Analyse abstraite

Analyse abstraite : ASTRÉE

▶ ASTRÉE

- ▶ Nov. 2003, prouve entièrement *automatiquement* l'*absence de toute erreur à l'exécution* (Run Time Errors) sur le logiciel de contrôle de vol primaire de l'Airbus A340 à commandes de vol électriques
 - ▶ 132.000 lignes de C
 - ▶ 1h20 sur un PC 32 bits 2,8 GHz (300 Mo de mémoire)
- ▶ Basée sur l'*interprétation abstraite*
 - ▶ *Approximations* faites sur la sémantique du langage C
 - ▶ Signale *toutes* les erreurs *possibles* (division par zéro, débordement de tableau, ...)
 - ▶ Parfois signale des erreurs qui n'en sont pas (*fausses alarmes*)



Analyse abstraite : outils

▶ Outils *commerciaux*

- ▶ Astrée : <http://www.absint.de/astree/>
 - ▶ Vérification des logiciels embarqués d'Airbus
- ▶ Polyspace : <http://www.mathworks.com/products/polyspace>
 - ▶ Issu des vérifications pour Ariane 502

▶ Outils *libres*

- ▶ Frama-C : <http://frama-c.com>
 - ▶ Framework plus général d'analyse et de preuve sur du code C



Analyse abstraite : aperçu

- ▶ Un *exemple* stupide : le signe d'un calcul
 - ▶ $\text{Sign}(x) = -1$ si $x < 0$, $+1$ sinon
 - ▶ $\text{Sign}(x * y) = \text{Sign}(x) * \text{Sign}(y)$ $\text{Sign}(x / y) = \text{Sign}(x) / \text{Sign}(y)$
 - ▶ $\text{Sign}(x + y) = ?$ $\text{Sign}(x - y) = ?$ \Rightarrow *approximations*
- ▶ Analyse abstraite
 - ▶ *Aller* de l'espace du programme dans un *espace abstrait*
 - ▶ Faire l'*analyse* : déterminer le signe d'une expression, nul ou pas, non dépassement des bornes d'un tableau, etc.
 - ▶ *Projeter* les résultats de l'analyse dans le programme initial
- ▶ *Fondement théorique* plus compliqué
 - ▶ Utilisation de fonctions monotones sur des treillis, de connections de Galois, ...
 - ▶ Garantir que l'analyse est *valide*
 - ▶ Tout ce qui est démontré dans l'abstraction l'est aussi sur le vrai système



Analyse abstraite : grille de lecture

- ▶ Domaines d'application / Problèmes possibles
 - ▶ *Code* Ada / C / C++. Vérifier les *fausses alarmes*
- ▶ Niveau d'expertise : *Nul*
- ▶ Niveau d'intervention :
 - ▶ sur le *code source final*, annotations
- ▶ Couverture du cycle de développement / Fidélité
 - ▶ Appliqué sur le code final, après chaque changement
- ▶ Disponibilité des outils / Niveau d'automatisme
 - ▶ Plusieurs outils disponibles, analyses *automatiques*
- ▶ Expressivité : qu'est-ce que je peux prouver ?
 - ▶ Certaines *classes* de propriétés : non division par zéro, accès hors bornes, dépassement de capacité, ...





Model checking

Model checking : SPIN

- ▶ Switch PathStar (Lucent Technologies)
 - ▶ *Vérification logique* du logiciel de gestion d'appel d'un switch commercial voix / données
 - ▶ Par ex. mise en attente, mode conférence, etc.
 - ▶ *Extraction de modèle* à partir du code ANSI-C original de l'application puis vérification sur le modèle
 - ▶ Vérification d'environ 80 propriétés écrites en *logique temporelle* linéaire
 - ▶ Un cluster de 16 processeurs utilisé pour faire les vérifications chaque nuit, pendant une période de plusieurs mois avant mise sur le marché
- ▶ Autres utilisations
 - ▶ Vérification d'algorithmes pour des missions spatiales
 - ▶ Deep Space I, Cassini, the Mars Exploration Rovers, Deep Impact, etc.



Model checking : aperçu

- ▶ Modèle d'un système avec *états* et *transitions gardées* : *automates*
- ▶ Vérifier des propriétés *temporelles* sur ce modèle
 - ▶ Utilisation d'une *logique temporelle* (\neq temps réel)
- ▶ *Plusieurs* logiques temporelles !
 - ▶ LTL (Linear Temporal Logics), CTL*, PLTL, MITL, ITL, AT, DC, DC*, ...
 - ▶ Différences : alternatives, temps quantifié, continu, dense
- ▶ Plusieurs façons de vérifier le modèle
 - ▶ *Énumération* des états : SPIN, Murphi, ...
 - ▶ *Symbolic* model checking: Lustre, Scade, NuSMV, ...
 - ▶ Considère simultanément un *ensemble d'états*
- ▶ Outils libres et commerciaux
 - ▶ Spin, SCADE Suite, NuSMV, Uppaal, ...



Exemples en logique temporelle

▶ *Logique booléenne* classique

▶ « \neg »: non, « \wedge »: et, « \vee »: ou, « \forall »: pour tout, « \exists »: il existe

▶ *Opérateurs temporels* : X, F, G, U, ...

▶ « XP » : P vérifiée au **prochain état** (neXt). Ex. : $P \wedge X(\neg P)$

▶ « FP » : P vérifiée dans le **Futur**

▶ « GP » : P vérifiée **Globalement**

▶ $G(\text{alert} \Rightarrow \text{F stop})$

▶ **À tout moment** (G), un état d'alerte est suivi par un état d'arrêt dans un état **futur** (F)

▶ « $P_1 U P_2$ » : P_1 est vérifiée **jusqu'à** (« **Until** ») ce que P_2 soit vérifiée

▶ $G(\text{alert} \Rightarrow (\text{alarm } U \text{ stop}))$

▶ **À tout moment** (G), une alerte déclenche **immédiatement** (\Rightarrow) une alarme **jusqu'à** (U) ce que l'état stop soit atteint



Model checking : exemple en SPIN

▶ SPIN : modélise des systèmes distribués et parallèles

▶ Processus, variables partagées et canaux de communication

```
mtype = { free, busy, idle, waiting, running };
```

```
show mtype h_state = idle;  
show mtype l_state = idle;  
show mtype mutex = free;
```

```
active proctype high() /* can run at any time */
```

```
{  
end: do  
  :: h_state = waiting;  
  atomic { mutex == free -> mutex = busy };  
  h_state = running;  
  
  /* critical section - consume data */  
  
  atomic { h_state = idle; mutex = free }  
od  
}
```

```
active proctype low() provided (h_state == idle) /* scheduling rule */
```

```
{  
end: do  
  :: l_state = waiting;  
  atomic { mutex == free -> mutex = busy};  
  l_state = running;  
  
  /* critical section - produce data */  
  
  atomic { l_state = idle; mutex = free }  
od  
}
```

Attendre

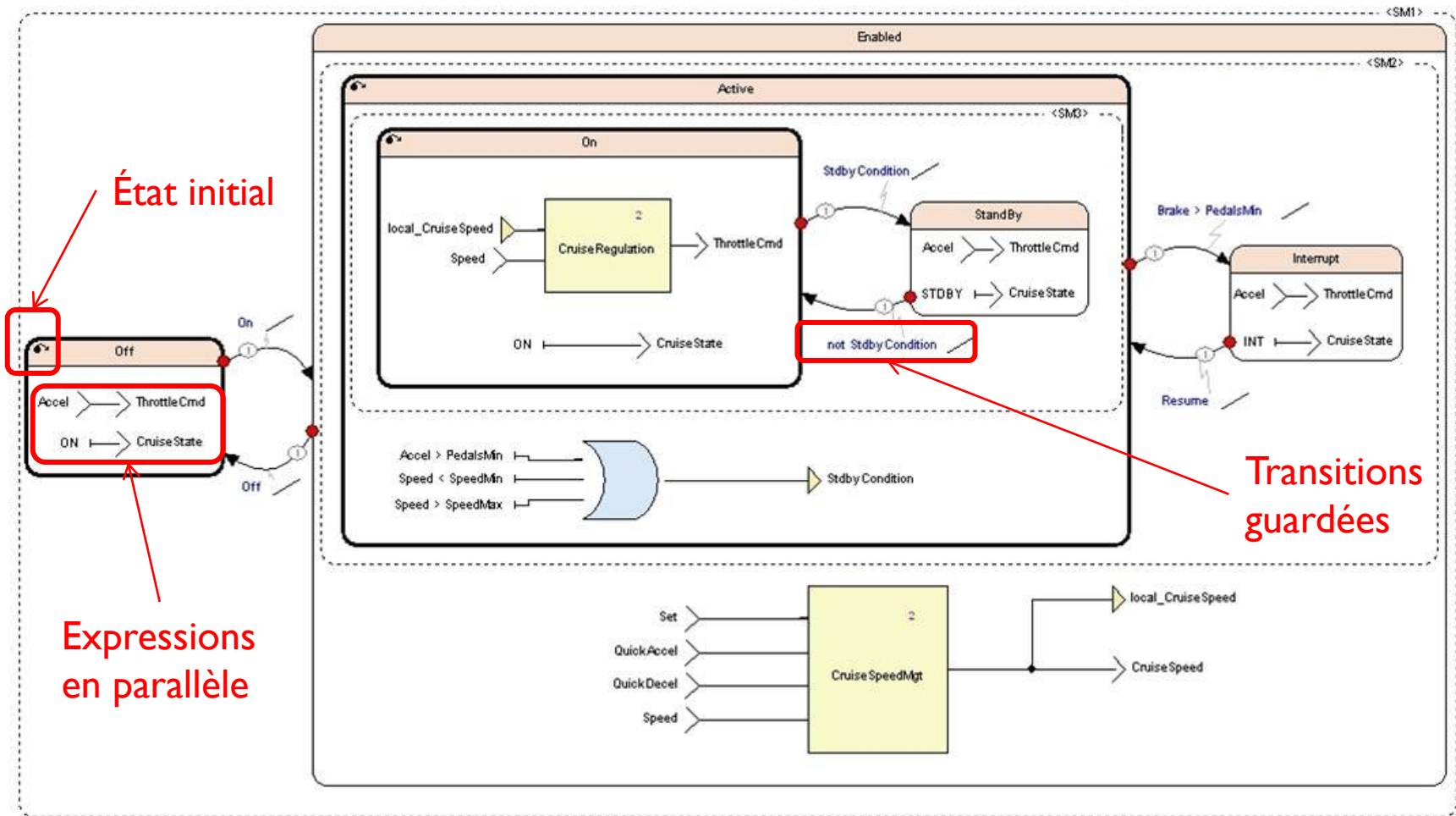
Faire
(avec entrelacement)

```
/*
```

- * Models the *Pathfinder scheduling algorithm* and explains the cause of the recurring reset problem during the mission on Mars
- * There is a high priority process, that consumes data produced by a low priority process.
- * Data consumption and production happens under the protection of a mutex lock.
- * The mutex lock conflicts with the scheduling priorities which can deadlock the system if high() starts up while low() has the lock set.
- * *There are 12 reachable states in the full (non-reduced) state space – two of which are deadlock states.*

- * Partial order reduction cannot be used here because of the 'provided' clause that models the process priorities.
- */

Model checking : SCADE Suite (A3xx)



État initial

Expressions en parallèle

Transitions gardées

Top Level of the Cruise Control application

Model checking : grille de lecture

- ▶ Domaines d'application / Problèmes possibles
 - ▶ *Matériel* (formules booléennes) et logiciels *concurrents*
 - ▶ Gérer l'*explosion des états*
- ▶ Niveau d'expertise : moyen
 - ▶ Écrire une *spécification formelle* mais *vérification automatique*
- ▶ Niveau d'intervention :
 - ▶ Sur un *modèle* du système, plutôt en phase de spécification
- ▶ Couverture du cycle de développement / Fidélité
 - ▶ *Modèle abstrait* en conception : lien manuel avec les spécifications
 - ▶ *Modèle extrait* du code final / *Code dérivé* du modèle (Scade)
- ▶ Disponibilité des outils / Niveau d'automatisme
 - ▶ Outils commerciaux et libres. Vérifications entièrement *automatiques*
- ▶ Expressivité : qu'est-ce que je peux prouver ?
 - ▶ Propriétés *temporelles*, violation d'*assertions*



Logique de Hoare

Logique de Hoare : la Méthode B

- ▶ Ligne de métro I4 sans conducteur à Paris
 - ▶ Environ 110.000 lignes de modèle B ont été écrites, générant environ 86.000 lignes de code Ada
 - ▶ 10 à 50 erreurs pour 1.000 ligne de code dans un logiciel classique
 - ▶ *Aucun bugs* trouvés après les preuves
 - ▶ *Ni* lors des tests d'*intégration*, des tests *fonctionnels*, des tests *sur site* ni depuis que la ligne est *en opération* (octobre 1998)
 - ▶ Le logiciel critique est toujours en *version 1.0*, sans bug détecté jusque là (en 2007)
- ▶ *Méthode B*
 - ▶ Construction de logiciels *corrects par construction*
 - ▶ Proposée par Jean-Raymond Abrial
 - ▶ *Raffiner* une spécification abstraite
 - ▶ En utilisant la logique de Hoare



Logique de Hoare : aperçu

- ▶ **Logique de Hoare** ou triplets de Hoare : $\{P\} C \{Q\}$
 - ▶ Règles logiques pour *raisonner* sur la correction d'un *programme* informatique
 - ▶ P : *Précondition*, C : *Commande*, Q : *Post-condition*
 - ▶ Si P est vraie, alors Q est vraie après exécution de la commande C

$$\overline{\{P\} \text{ skip } \{P\}} \quad \overline{\{P[E/x]\} x := E \{P\}} \quad \{x + 1 = 43\} y := x + 1 \{y = 43\}$$

$$\frac{\{P\} S \{Q\}, \{Q\} T \{R\}}{\{P\} S; T \{R\}} \quad \frac{\{B \wedge P\} S \{Q\}, \{\neg B \wedge P\} T \{Q\}}{\{P\} \text{ if } B \text{ then } S \text{ else } T \text{ endif } \{Q\}} \quad \frac{P' \rightarrow P, \{P\} S \{Q\}, Q \rightarrow Q'}{\{P'\} S \{Q'\}}$$

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{ while } B \text{ do } S \text{ done } \{\neg B \wedge P\}}$$

- ▶ Calcul de P : calcul de la plus faible pré-condition
- ▶ La *terminaison* doit aussi être prouvée
- ▶ Preuves statiques : vraies pour *toutes les exécutions* !



Logique de Hoare : exemple Framac-C

- ▶ Framac-C / Jessie : logique de Hoare sur du vrai code C
 - ▶ Utilisation de *démonstrateurs automatiques* (SMT solvers) pour les preuves en utilisant *Why*

```
/*@ requires n >= 0 && \valid_range(t,0,n-1); */
int binary_search(long t[], int n, long v) {
    int l = 0, u = n-1;
    /*@ loop invariant
       @ 0 <= l && u <= n-1;
       @ loop variant u-l;
    @*/
    while (l <= u) {
        int m = (l + u) / 2; // ← notre bug
        if (t[m] < v) l = m + 1;
        else if (t[m] > v) u = m - 1;
        else return m;
    }
    return -1;
}
```



Note sur les SMT solvers

- ▶ Satisfiability Modulo Theories (SMT) solver
 - ▶ *Satisfiability*: vérifier qu'une formule peut être *satisfaite*
 - ▶ *Modulo Theory* : *présupposés* sur l'arithmétique, les tableaux, les types de données algébriques, ...
 - ▶ En d'autres termes : vérifier que l'on peut *trouver une solution* à une formule booléenne
- ▶ Outils *entièrement automatiques*
 - ▶ Vérifie ou échoue (erreur ou timeout)
 - ▶ Z3, CVC3, Alt-Ergo, Yice, ...

(* first-order logic *)

type t

logic c : t

logic f : t -> t

logic p,q : t -> prop

goal fol_1 : ($\forall x:t. p(x)$) \Rightarrow p(c)

goal fol_2 : ($\forall x:t. p(x) \Leftrightarrow q(x)$) \Rightarrow p(c) \Rightarrow q(c)

(* equality *)

goal eq_1 : p(c) \Rightarrow $\forall x:t. x=c \Rightarrow p(x)$

(* arithmetic *)

goal arith_1 : $\forall x:int. x=0 \Rightarrow x+1=1$

goal arith_2 : $\forall x:int. x < 3 \Rightarrow x \leq 2$

(* propositional logic *)

logic A,B,C : prop

goal prop_1 : A \Rightarrow A

goal prop_2 : (A or B) \Rightarrow (B or A)



Exemple Frama-C : correction

▶ *Correction*

- ▶ montrer que le programme a un comportement correct

▶ *Spécifier* le comportement attendu

▶ *Prouver* que la spécification est satisfaite

▶ *Invariant de boucle*

- ▶ Construction *progressive* de la propriété requise

```
/*@ predicate array_sorted{L}(long *t, integer a, integer b) =  
  @ \forall integer i,j; a <= i <= j <= b ==> t[i] <= t[j];  
  @*/
```

```
/*@ requires n >= 0 && \valid_range(t,0,n-1);  
  @ ensures -1 <= \result < n;
```

```
@ behavior success:  
  @ ensures \result >= 0 ==> t[\result] == v;
```

```
@ behavior failure:  
  @ assumes array_sorted(t,0,n-1);  
  @ ensures \result == -1 ==>  
  @ \forall integer k; 0 <= k < n ==> t[k] != v;  
  @*/
```

```
int binary_search(long t[], int n, long v) {  
  int l = 0, u = n-1;
```

```
  /*@ loop invariant  
  @ 0 <= l && u <= n-1;
```

```
  @ for failure:  
  @ 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) / 2;  
    int m = l + (u - l) / 2;  
    //@ assert l <= m <= u;  
    if (t[m] < v) l = m + 1;  
    else if (t[m] > v) u = m - 1;  
    else return m;  
  }  
  return -1;  
}
```

0	1	...	l	...	k	...	u	...	n-1
!=v	!=v	!=v	?	?	?v	?	?	!=v	!=v

Exemple Framac : preuves automatiques

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "View", "Tools", and "Help".

Context:

- Unproved goals
- All goals

Provers:

- Alt-Ergo 0.93
- Coq 8.2pl2
- CVC3 2.2
- Simplify 1.5.4
- Z3 2.19

Transformations:

- Split
- Inline

Tools:

- Edit
- Replay

Cleaning:

- Remove
- Clean

Proof monitoring:

- Waiting: 0
- Scheduled: 0
- Running: 0
- Interrupt

Theories/Goals Table:

Theories/Goals	Status	Time
binary-search.mlw	✓	
Jessie_model	✓	
WP Jessie_program	✓	
Function binary_search, default behavior	✓	
Alt-Ergo 0.93	✓	0.09
Function binary_search, Behavior 'failure'	✓	
Alt-Ergo 0.93	✓	0.56
Function binary_search, Behavior 'success'	✓	
Alt-Ergo 0.93	✓	0.01
Function binary_search, Safety	✓	
Alt-Ergo 0.93	✓	0.58

Code Editor:

```
14 int binary_search(long t[], int n, long v) {
15   int l = 0, u = n - 1;
16   /*@ loop invariant
17    @ 0 <= l && u <= n - 1;
18    @ for failure:
19    @ loop invariant
20    @ \forall integer k; 0 <= k < n && t[k] == v ==> l <= k <= u;
21    @ loop variant u - l;
22    @*/
23   while (l <= u) {
24     // int m = (l + u) / 2;
25     int m = l + (u - l) / 2;
26     /*@ assert l <= m <= u;
27     if (t[m] < v) l = m + 1;
28     else if (t[m] > v) u = m - 1;
29     else return m;
30   }
```

File: /home/david/2011-11-29-ESIR3-presentation/c-binary-search/binary-s

Exemple d'outil : Atelier B

The screenshot displays the Atelier B software interface. The main window shows a project workspace with a tree view on the left containing components like 'fuel_tank' and 'test'. A table in the center lists components and their status:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
switch	OK	OK	0	0	0	-
switch_impl...	OK	OK	11	0	0	OK

An inset window titled 'switch - Atelier B' shows the B code for the 'switch' component:

```
/* switch
 * Author: Mentre
 * Creation date: jeu. oct. 8 2009
 */
MACHINE
  switch
SETS
  POSITION = {normal,reverse,void}

OPERATIONS
pos <-- estimate(m1,m2,m3) =
  PRE
    m1: POSITION &
    m2: POSITION &
    m3: POSITION
  THEN
    SELECT
      normal: {m1,m2,m3} &
      reverse /: {m1,m2,m3}
    THEN
      pos:=normal
  WHEN
    reverse: {m1,m2,m3} &
    normal /: {m1,m2,m3}
  THEN
    pos:=reverse
  ELSE
    pos:=void
  END
END
END
```

The inset window also features a 'B Symbols' table and an 'Outline' view.

Description	Ascii
Functions and relations	
→	Partial function
→	Total function
→	Partial surjection
→	Surjection
λ	Lambda function
→	Override
↔	Relation
←	Domain abstraction
←	Domain restriction
↔	Partial injection
↔	Total injection
↔	Partial bijection
↔	Total bijection
⊗	Direct product
R*	Reflexive closure
R*	Closure

The 'Outline' view shows the structure of the 'switch' component:

- MACHINE
- switch
- SETS
- POSITION
- OPERATIONS
- estimate

The status bar at the bottom indicates 'Line: 1 Column: 1 errors'.

Logique de Hoare : grille de lecture

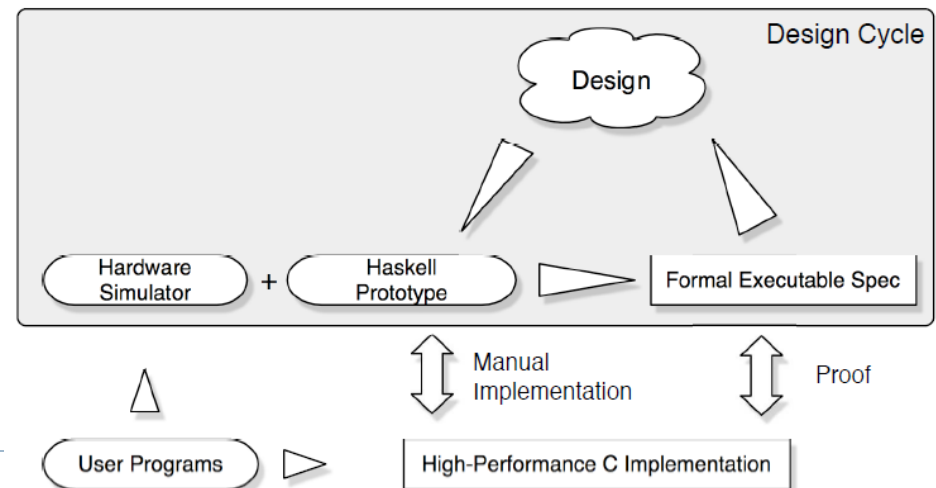
- ▶ Domaines d'application / Problèmes possibles
 - ▶ Convient à *tous types* de programmes
 - ▶ *Parfois difficile* d'exprimer les assertions logiques (par ex. boucles)
- ▶ Niveau d'expertise : moyen à élevé / Niveau d'intervention
 - ▶ Écrire une *spéc formelle* puis vérification *automatique* ou *manuelle*
- ▶ Couverture du cycle de développement / Fidélité
 - ▶ Sur du *code final* (ex. Framac, SPARK Ada)
 - ▶ *Modèle formel dérivé* en code final (Méthode B)
- ▶ Disponibilité des outils / Niveau d'automatisme
 - ▶ Outils commerciaux et libres. Automatique et manuel
- ▶ Expressivité : qu'est-ce que je peux prouver ?
 - ▶ Toutes *propriétés sur les états* (pas de propriétés temporelles)



Démonstrateurs interactifs de théorèmes

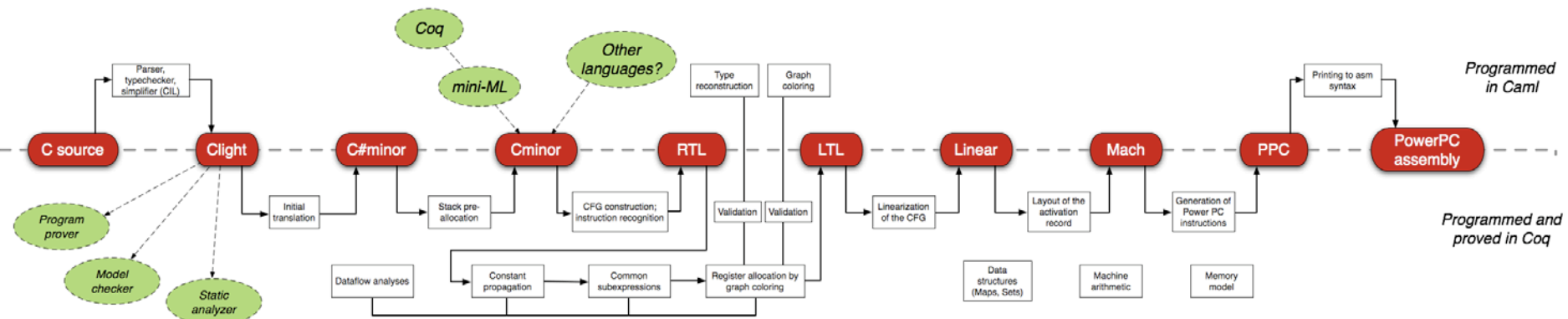
Démonstrateurs interactifs : seL4

- ▶ seL4 : micro-noyau
- ▶ Définition formelle d'un *spécification abstraite*
 - ▶ Signification de la correction du noyau
 - ▶ Description de ce que fait le micro-noyau pour chaque entrée (trap instruction, interruption, ...)
 - ▶ Mais pas nécessairement *comment* c'est fait
- ▶ *Preuve* mathématique que la *réalisation en C* correspond toujours à la spécification
 - ▶ Dans l'assistant de preuve Isabelle/HOL



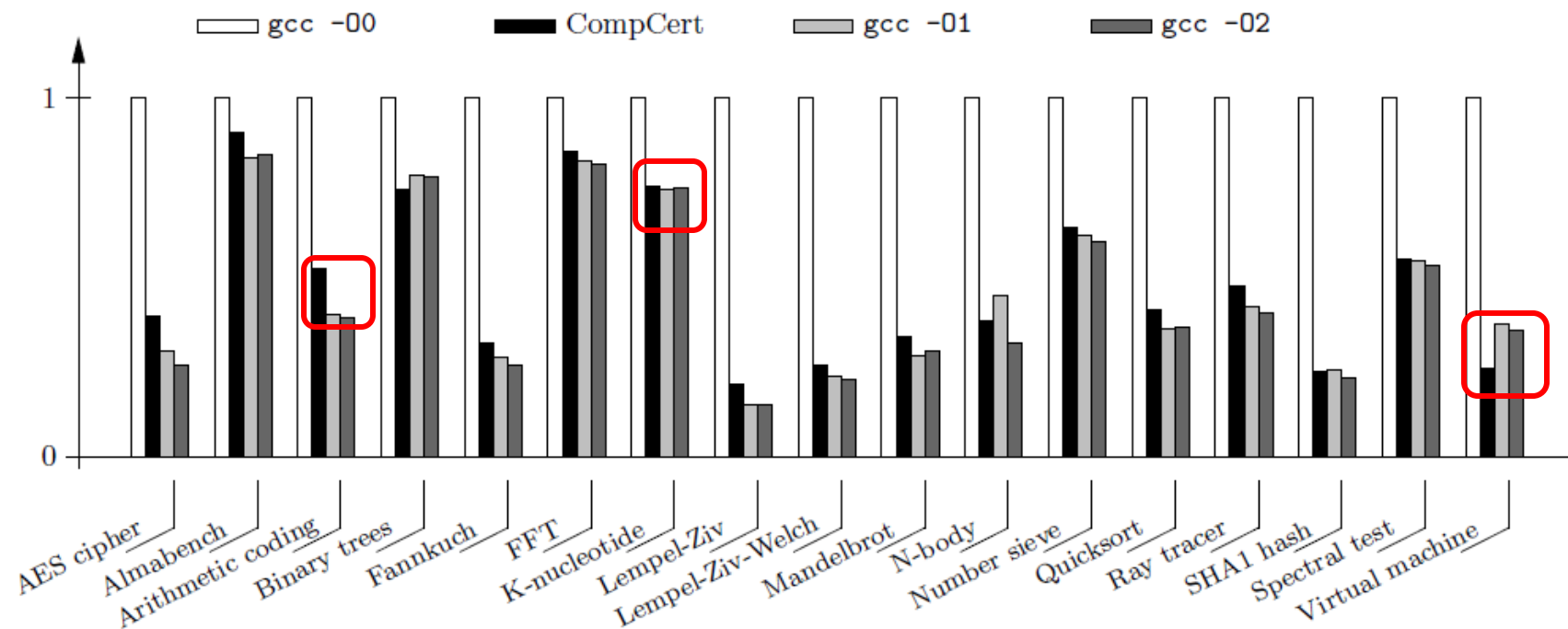
Démonstrateurs interactifs : Compcert

- ▶ Compcert : un *compilateur C* certifié <http://compcert.inria.fr/>
 - ▶ Génère de l'assembleur PowerPC et ARM à partir de Clight, un large sous-ensemble du langage C
 - ▶ Principalement écrit dans le langage de l'*assistant de preuve Coq*
 - ▶ Sa *correction* a été *entièrement prouvée* dans Coq
 - ▶ Correction = assembleur généré est *sémantiquement équivalent* au source initial



Compcert : performances

- ▶ **Performances** de compcert proche de gcc -O1
 - ▶ Correction ne veut **pas** dire mauvaises performances !



Démonstrateurs interactifs : aperçu

- ▶ Outils pour faire un raisonnement proche du *raisonnement mathématique*
 - ▶ Coq, PVS, Isabelle/HOL, ... (assistants de preuves)
- ▶ Permet d'exprimer des *propriétés complexes*
 - ▶ Ex. : vérifier les règles de preuve de l'Atelier B (Méthode B) dans Coq (travail BiCoq)
- ▶ La logique utilisée est *non décidable*
 - ▶ Donc l'*utilisateur* est nécessaire pour faire les preuves



Utilité de la logique constructive

- ▶ Certains assistants de preuve utilisent une logique *constructive* (par ex. Coq)
 - ▶ Une preuve montre comment construire l'objet mathématique recherché (en plus de dire qu'il existe)
- ▶ Approche générale : *isomorphisme de Curry-Howard*
 - ▶ Une preuve est un programme, la formule qu'elle prouve est un type pour ce programme

$\lambda x. x+1 : \text{int} \rightarrow \text{int}$
terme type

λ -calcul	Logique	Programmation
Type	Formule	Spécification
Terme	Preuve	Programme

- ▶ On peut *extraire* un programme d'une preuve
-



Démonstrateurs interactifs : grille de lecture

- ▶ Domaines d'application / Problèmes possibles
 - ▶ Convient à *tous* types de domaine. Haut niveau de confiance
 - ▶ Parfois *difficile* à utiliser
- ▶ Niveau d'expertise : élevé / Niveau d'intervention
 - ▶ Besoin de tout spécifier et de prouver dans l'assistant de preuve
 - ▶ Utilisable du plus *abstrait* (logique) au plus *concret* (compilateur C, ...)
- ▶ Couverture du cycle de développement / Fidélité
 - ▶ Cible principalement les *modèles formels* et leurs raisonnements
 - ▶ Utilisations *produits* : compcert, seL4, JavaCard chez Gemalto
- ▶ Disponibilité des outils / Niveau d'automatisme
 - ▶ Beaucoup d'outils libres. Usage principalement *manuel*
- ▶ Expressivité : qu'est-ce que je peux prouver ?
 - ▶ Prouver *tous* types de propriétés





Pour finir

Formalismes omis

- ▶ Il y a *beaucoup* d'approches et outils formels !
 - ▶ Cette présentation n'est pas exhaustive
- ▶ Langages de spécification *algébriques* et *ensemblistes*
 - ▶ Z, VDM, Alloy, ACT-ONE, CLEAR, OBJ, ...
- ▶ *Algèbres concurrentes* et autres formalismes *concurrents*
 - ▶ CSP, CSS, Process Algebra, π -calculus, ...
 - ▶ Lotos, Petri Nets, Unity, Event B, TLA+, langages synchrones, ...
- ▶ Systèmes de *types*
 - ▶ Utilisés dans certains *langages* de programmation comme *OCaml*, *Haskell*, ...
 - ▶ Permettent d'*éviter* certaines *classes* de bugs (si bien utilisés)
 - ▶ Par ex., conserver la structure de structures de données, éviter de mélanger des entiers ayant un sens différent, ...



Conclusion (1 / 2)

▶ Méthodes formelles

- ▶ Un excellent moyen pour *améliorer* la *qualité* du matériel et logiciel
- ▶ Pas la réponse à tout mais une bonne réponse
 - ▶ Pas seulement pour des systèmes critiques !

▶ Beaucoup d'approches !

- ▶ Approches *principales* : Interprétation abstraite, Model checking, Logique de Hoare et Démonstrateurs interactifs
- ▶ Certaines sont *faciles*, d'autres plus *difficiles*
 - ▶ De entièrement automatiques à entièrement manuelles
- ▶ *Utiles* mêmes si elles ne sont pas complètement employées
 - ▶ Même une seule spécification formelle est utile !
 - IBM CISC information system update : -9% coûts dev., ÷ 2.5 bugs



Conclusion (2/2)

- ▶ *Intégrez* les méthodes formelles dans vos développements
 - ▶ Comme les tests, un *outil* de plus
 - ▶ *Obligatoires* dans certains domaines : ferroviaire, aéronautique, ...
 - ▶ Avoir un *expert* sous la main pour bien les utiliser
 - ▶ Projets libres, conseils sur les listes de diffusion !
 - ▶ Beaucoup d'*outils* sont *disponibles*
 - ▶ Dont la plupart sont Libres et/ou gratuits
- ▶ Posez-moi des *questions* : dmentre@linux-france.org
- ▶ Dans un futur (si ?) lointain
 - ▶ « We envision a world in which computer programmers make no more mistakes than other professionals, a world in which *computer programs are always the most reliable components* of any system or device. »
 - C.A.R. Hoare et al., Verified Software Initiative Manifesto





Backup slides

Licence de cette présentation

- ▶ Cette présentation est sous licence Art Libre 1.3
 - ▶ <http://artlibre.org/licence/lal>
 - ▶ Copyright 2011 David MENTRÉ
 - ▶ Exception : la plupart des images, propriétés de leur auteur

