Introduction aux méthodes formelles

David MENTRÉ – dmentre@linux-france.org

2016

Plan de la présentation

- Aperçu général
  - *Pourquoi* utiliser les méthodes formelles et comment ?

1. Interprétation abstraite
   - Démonstration *automatique* de propriétés sur du code réel
2. Programmation par contrat
   - Prouver des propriétés génériques *sur un programme* complet
3. Programmation par raffinement
   - Produire un programme *correct par construction*
4. Model checking
   - Vérifier des propriétés *temporelles* sur des automates
5. Programmation certifiée
   - Toute la *puissance des mathématiques* … et sa complexité

- Conclusion
  - Ce qu'il faut en *retenir*

Prélude : deux exemples illustratifs

Ex1 : ce code contient-il 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;
  }
```

Ex1 : ce code contient-il 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 $2^{31}-1$

Ex2 : ce code contient-il une erreur ?

- 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.
    }
    ```


    ```

Ex2 : ce code contient-il une erreur ?

- 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.
}
```

dépassement si low + high > 2^{31}-1

Solution

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

Impact

Bug présent de le *JDK* de ~~Sun~~Oracle ! Il a impacté des utilisateurs http://googleresearch.blogspot.com/2006/06/ext a-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* et *précision* des mathématiques !
    - A et B ou C : $(A \wedge B) \vee C$ *vs.* $A \wedge (B \vee C)$?
    - $a < 3$ *vs.* $a \leq 3$?

- 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 !)
  - Spécifications algébriques, Machines d'état abstraites, Interprétation abstraite, *Model Checking*, Systèmes de types, Démonstrateurs de théorèmes automatiques ou interactifs, …
  - Nous en présenterons certains par des *exemples* et *démos*

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'expliciter 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 : *Therac 25 (1985-1987, 5 morts), Pentium FDIV bug (Jan. 1995, pre-tax charge of $475 million for Intel), Ariane 501 (1996, ~$370 million), Toyota Unintended Acceleration (2007-2013, plusieurs morts, ~ $1200 million), …*

Peut-on utiliser les méthodes formelles ?

- La *maturité* des outils et leurs *usages* s'est beaucoup amélioré ces dernières années
  - Outils de plus en plus automatiques
  - Pas besoin d'avoir une thèse pour les utiliser
    - Exemple : ingénieurs de ClearSy (Méthode B)
  - Mais certains sont toujours complexes

	Alt-Ergo 0.95.1	Alt-Ergo 0.95.2	Alt-Ergo mast. bran. *	Ctrl-Alt-Ergo mast. branch *
Release date	Mar. 05, 2013	Sep. 20, 2013	- - -	- - -
Why3 benchs	2.270 (92 %)	**2.288 (93 %)**	2.308 (93 %)	**2.363 (96 %)**
SPARK benchs	2.351 (74 %)	**2.360 (75 %)**	2.373 (75 %)	**2.404 (76 %)**
BWare benchs	5.609 (53 %)	**9.437 (89 %)**	10.072 (95 %)	**10.373 (98 %)**

- La plupart des outils sont *disponibles gratuitement* et/ou *librement*
  - Permet des expérimentations à coût raisonnable
  - http://gulliver.eu.org/free_software_for_formal_verification
  - http://www.atelierb.eu/telecharger-latelier-b/
- Donc *oui* !

Comment les appliquer ?

- Quel est votre *problème* ? Que voulez-vous *garantir* ?
  - Trouver une méthode formelle qui corresponde
    - À votre domaine d'application
    - À vos problèmes
    - À vos contraintes de temps et de coûts
    - À votre formation en informatique
    - …
- Comment les mettre en œuvre ?
  - Les *intégrer* à votre *cycle de développement*
    - Retour à *moyen/long* terme
      - Ex : mettre à jour la spécification formelle quand celle-ci ou le système réalisé changent
    - Retour lors des *tests*, des *mises à jour*
  - 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 ?

Cas Ariane 501

Histoire d’un bug logiciel

Utile le formel ? (1/3)

- Le formel est-il 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.

Mauvaises données

- 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.

Exception à l'exécution

- 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.

Utile le formel ? (3/3)

- Chaîne d'événements techniques

[…]

Cause informatique de l'exception: integer overflow

- 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.

Pourquoi l'erreur s'est propagée

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* (implicitement) !

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)

- Mais le formel ne résout *pas* tout (cf. problème de spéc.)

Interprétation abstraite

Interprétation 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)
    - Janvier 2004 : analyse étendue à l'*A380*

- Basée sur l'*interprétation abstraite*
  - *Approximation* de la sémantique du langage C
    - Signale *toutes* les erreurs *possibles* (division par zéro, débordement de tableau, overflow)
    - Mais peut parfois signaler des erreurs qui n'en sont pas (*fausses alarmes*)

Interprétation abstraite : outils

- Outils *propriétaires*
  - *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/*Value analysis* : http://frama-c.com
    - Framework plus général d'analyse et de preuve sur du code C
    - *Démo* : frama-c-gui -val abs-value.c

Interprétation abstraite : aperçu

- Un *exemple* stupide : le signe d'un calcul
  - Sign(x) = -1 si $x < 0$, +1 sinon
  - Sign($x * y$) = Sign(x) * Sign(y) Sign(x / y) = Sign(x) / Sign(y)
  - Sign($x + y$) = *?* Sign($x - y$) = *?* ⇒ *approximations*

- Interprétation 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*
  - 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

Interprétation abstraite : grille de lecture

- Domaines d'application / Problèmes possibles
  - *Code* Ada / C / C++ / Java
  - Vérifier les *alarmes*
- Niveau d'expertise : *faible*
- 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
  - Outils *libres* et *propriétaires* 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, …

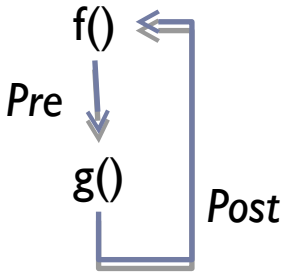
Programmation par contrat

Programmation par contrat : Frama-C

- **Frama-C** : framework pour l'analyse et la preuve de code C
  - Construction d'un arbre de syntaxe abstrait à partir de fichiers C
  - Applications de diverses analyses : plug-ins
    - WP et Jessie : preuve en logique de Hoare
  - Développé par le CEA et l'INRIA / Libre / http://frama-c.com
  - Utilisations : Airbus, Dassault Aviation, IAE Brasil, …

frama C
Software Analyzers

- Contrat sur chaque fonction
  - Pré-condition : conditions vérifiées par l'*appelant*
  - Post-condition : conditions garanties par l'*appelé*



- Popularisé par Eiffel (Betrand Meyer)
  - Maintenant disponible en Ada 2012, C (Frama-C), C#, Visual Basic (Microsoft Code Contracts for .NET), Dafny http://rise4fun.com/Dafny/

Eiffel Software

- Contrat vérifié *statiquement* (preuve) ou *dynamiquement* (test)
  - Ou *mélange* des deux : SPARK 2014, plug-in E-ACSL dans Frama-C, …

abs() avec un contrat formel

- Fonction prouvée *correcte* dans tous les cas *légaux* d'utilisation
  - *Démo !* : frama-c-gui -wp abs-proved.c

```
/*@ requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
 */
int abs(int x){
  if (x < 0)
    return -x;
  else
    return x;
}
```

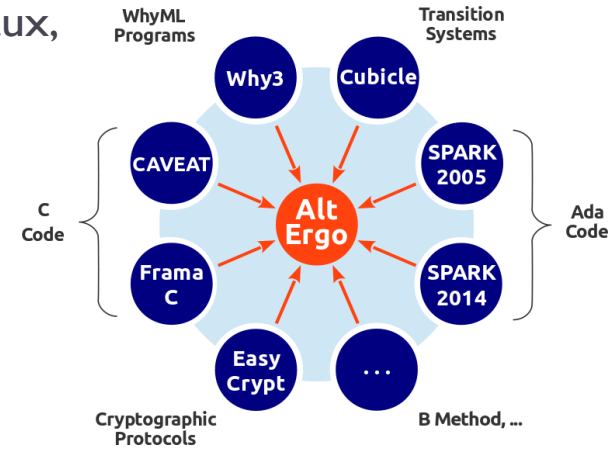
Condition d'***entrée*** : pour éviter les overflow

Conditions de ***sortie*** : résultat toujours positif et valeur absolue

- Utilisation sous-jacente de *démonstrateurs automatiques*
  - Notamment *SMT solvers*
  - Démonstrateurs *spécialisés*
    - Méthode B (Atelier B), réels/flottants (Gappa, MetiTarski), …

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 inégalités, 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)
  - *Alt-Ergo*, Z3, CVC3, Yice, …
    - Alt-Ergo *certifié* DO-178B/C (Airbus et Dassault)

(* *propositional logic* *)
logic A,B,C : prop
goal prop_1 : A ⇒ A
goal prop_2 : (A or B) ⇒ (B or A)

(* *equality* *)
goal eq_1 : p(c) ⇒ ∀x:t. x=c ⇒ p(x)

(* *arithmetic* *)
goal arith_1 : ∀ x:int. x=0 ⇒ x+1=1
goal arith_2 : ∀ x:int. x < 3 ⇒ x <= 2

(* *first-order logic* *)
type t

logic c : t
logic f : t -> t
logic p,q : t -> prop

goal fol_1 : (∀ x:t. p(x)) ⇒ p(c)
goal fol_2 : (∀ x:t. p(x) ⇔ q(x)) ⇒ p(c) ⇒ q(c)

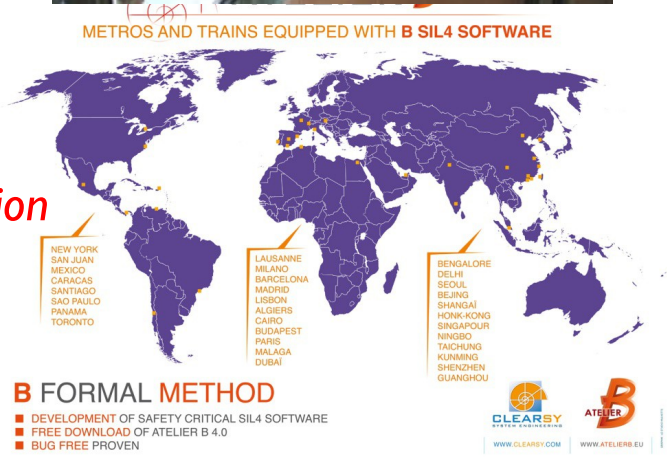
Prog. par contrat : grille de lecture

- Domaines d'application / Problèmes possibles
  - Code *Ada* (SPARK 2014), *C* (Frama-C), *C#* (MS Code Contracts), *WhyML* (Why3), …
  - *Parfois difficile* d'exprimer les assertions logiques (par ex. invariants de boucles)
- Niveau d'expertise : *moyen à fort*
- 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
  - Outils *libres* et *propriétaires* disponibles
  - Annotations *manuelles* (x2 sur le code), analyses *automatiques*
- Expressivité : qu'est-ce que je peux prouver ?
  - *Tout* type de propriété sur les *états*
    - Mais *limitation* des prouveurs automatiques (ex. induction)

Programmation par raffinement

Prog. par raffinement : Méthode B

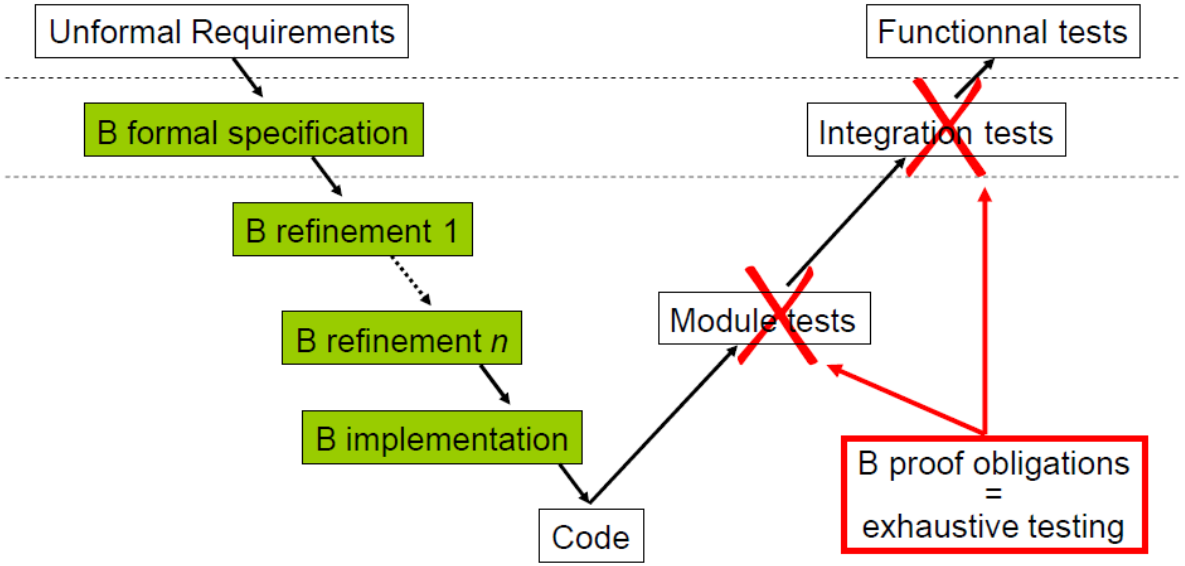
- Ligne de métro 14 sans conducteur à Paris
  - Environ 110 000 lignes de modèle B ont été écrites, générant environ 86 000 lignes de code Ada
    - NB : 10 à 50 erreurs pour 1000 lignes de code dans un logiciel classique
  - *Aucun bug* trouvé après les preuves
    - *Ni* lors des tests d'*intégration*, 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



Méthode B : principe

- Partir d'une spécification de *haut niveau*
  - Spécification *descriptive*
- *Ajouter des détails* jusqu'à arriver à un programme exécutable
  - Préciser l'*algorithme*
  - Préciser les *structures de données*

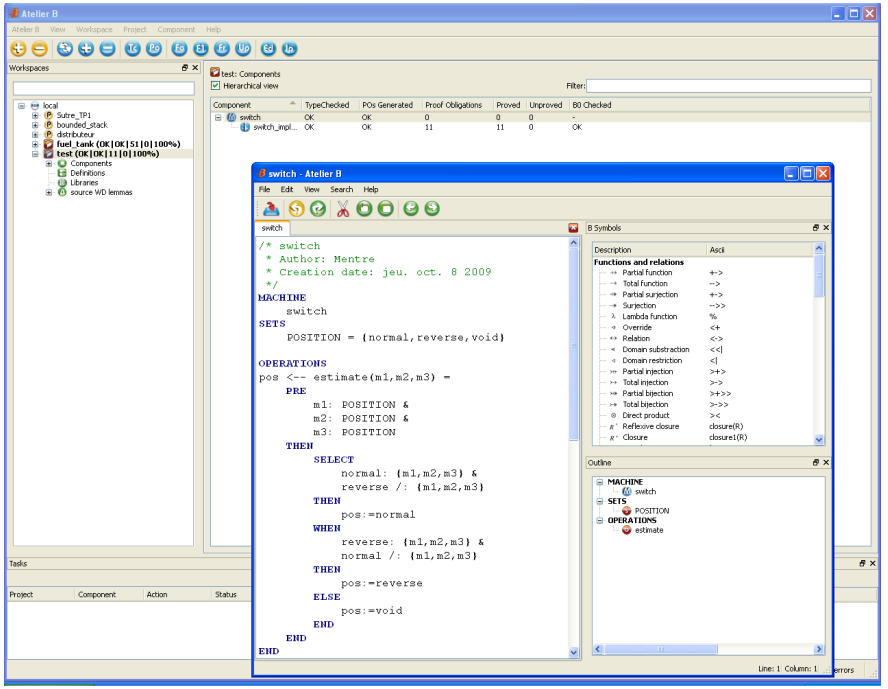
- *Avantages*
  - *Substituer* des preuves aux tests
    - *Exhaustivité* !
    - Coûts *réduits* !
  - Logiciel *correct par construction*
  - *Maintenance* facilitée



Exemple : Atelier B sur un aiguillage

- Aiguillage à trois positions : normal, reverse, void (inconnu)
- Programme : donner la configuration de l'aiguillage
  - Trois détecteurs : m1, m2 et m3
  - Si pas de contradiction (normal et reverse simultanément) : position lue
  - Sinon : void

- *Démo* : Atelier B
  - Spécification
  - Implémentation
  - Preuves : implémentation conforme à la spécification



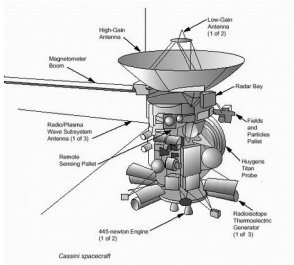
Prog. par raffinement : grille de lecture

- Domaines d'application / Problèmes possibles
  - Convient à *tous types* de programmes
  - *Parfois difficile* d'exprimer les assertions logiques (par ex. boucles)
  - Notation / formalisme à *acquérir*
- Niveau d'expertise : moyen à élevé / Niveau d'intervention
  - Écrire une *spéc formelle* puis vérification *automatique* ou *manuelle*
  - Génération *automatique* du code exécutable (C, Ada, Ladder, …)
- Couverture du cycle de développement / Fidélité
  - *Modèle formel dérivé* en code final, *correct par construction*
  - Processus de développement à adapter
- Disponibilité des outils / Niveau d'automatisme
  - Outils commerciaux
  - *Automatique* et *manuel* (manuel possible si automatique échoue)
- Expressivité : qu'est-ce que je peux prouver ?
  - *Tout* type de propriétés sur les *états* (pas de propriétés temporelles)

Model checking

Model checking : SPIN

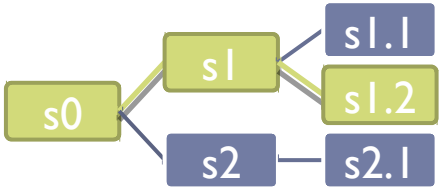
- Switch PathStar (Lucent Technologies)
  - *Vérification logique* du logiciel de gestion d'appel d'un switch commercial voix / données
    - Par ex. mise en attente, mode conférence, etc.
  - *Extraction de modèle* à partir du code ANSI-C original de l'application puis vérification sur le modèle
  - 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



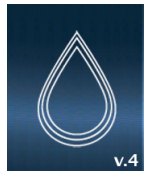
- Autres utilisations
  - Vérification d'algorithmes pour des missions *spatiales*
    - Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc.
  - Contrôle de *barrières anti-inondation* (Pays-Bas), Enquête *véhicule* Toyota (USA), Vérification protocole ISO/IEEE 11073-20601 pour appareils *médicaux*

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* (≠ temps réel)
- Il existe *plusieurs* logiques temporelles !
  - LTL (Linear Temporal Logic), 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 Design Verifier, NuSMV, …
    - Considère simultanément un *ensemble d'états*
- Outils *libres*, *gratuits* et *commerciaux*
  - Spin, NuSMV, Uppaal, SCADE Design Verifier, …



Exemples en logique temporelle

- *Logique booléenne* classique
  - « ¬ »: non, « ∧ »: et, « ∨ »: ou, « ∀ »: pour tout, « ∃ »: il existe
- *Opérateurs temporels* : X, F, G, U, …
  - « *X*P » : P vérifiée au prochain état (ne*X*t). Ex. : P ∧ X(¬P)
  - « *F*P » : P vérifiée dans le *F*utur
  - « *G*P » : P vérifiée *G*lobalement
    - G(alert ⇒ 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* (« *U*ntil ») ce que P_2 soit vérifiée
    - G(alert ⇒ (alarm U stop))
    - *À tout moment* (G), une alerte déclenche *immédiatement* (⇒) 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,waiting
show mtype l_state = idle,waiting running
show mtype mutex = free,busy

active proctype high()  /* can run at any time */
{
end:   do
4:     :: 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
1:     :: l_state = waiting;
2:        atomic { mutex == free -> mutex = busy};
3:        l_state = running;

          /* critical section - produce data */

          atomic { l_state = idle; mutex = free }
       od
}
```

Attendre

Faire
(avec entrelacement)

Parce que mutex != free

Parce que h_state != idle

```
/*
 * 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.
 */
```

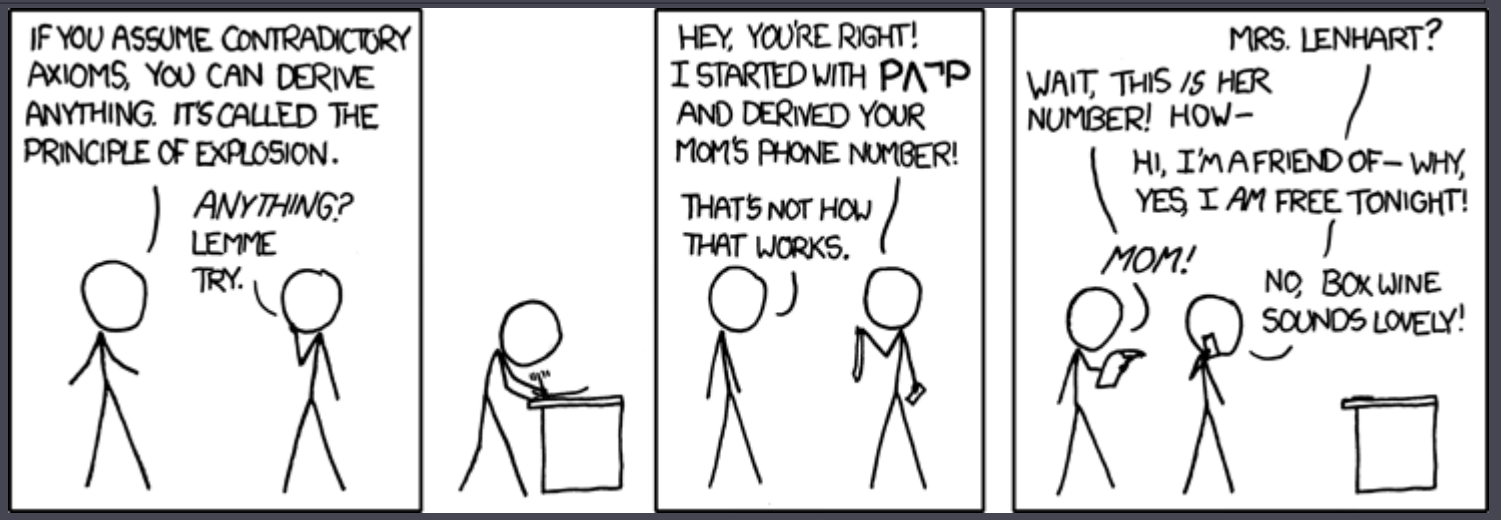

```
1:    proc  1 (low) line  35 "pathfinder" (state 1) [l_state = waiting]
2:    proc  1 (low) line  35 "pathfinder" (state 2) [((mutex==free))]
2:    proc  1 (low) line  35 "pathfinder" (state 3) [mutex = busy]
3:    proc  1 (low) line  36 "pathfinder" (state 5) [l_state = running]
4:    proc  0 (high) line  27 "pathfinder" (state 1) [h_state = waiting]
spin: trail ends after 4 steps
```

Deadlock !

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 (Spin/Modex) / *Code dérivé* du modèle (SCADE)
- Disponibilité des outils / Niveau d'automatisme
  - Outils *commerciaux* et *libres*
  - Vérifications entièrement *automatiques*
  - Production d'un *contre exemple* en cas d'erreur
- Expressivité : qu'est-ce que je peux prouver ?
  - Propriétés *temporelles*, violation d'*assertions*

Programmation certifiée

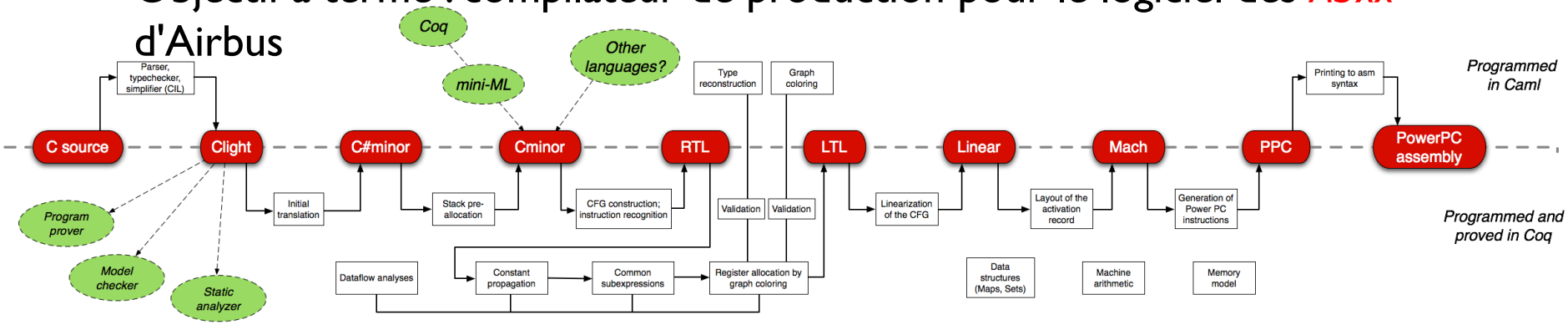


CompCert

- CompCert : un *compilateur C* certifié http://compcert.inria.fr/
  - Génère de l'assembleur PowerPC, ARM et x86 à partir de C90
  - Principalement écrit dans le langage de l'*assistant de preuve Coq*
  - Le compilateur est *extrait* de sa preuve de correction

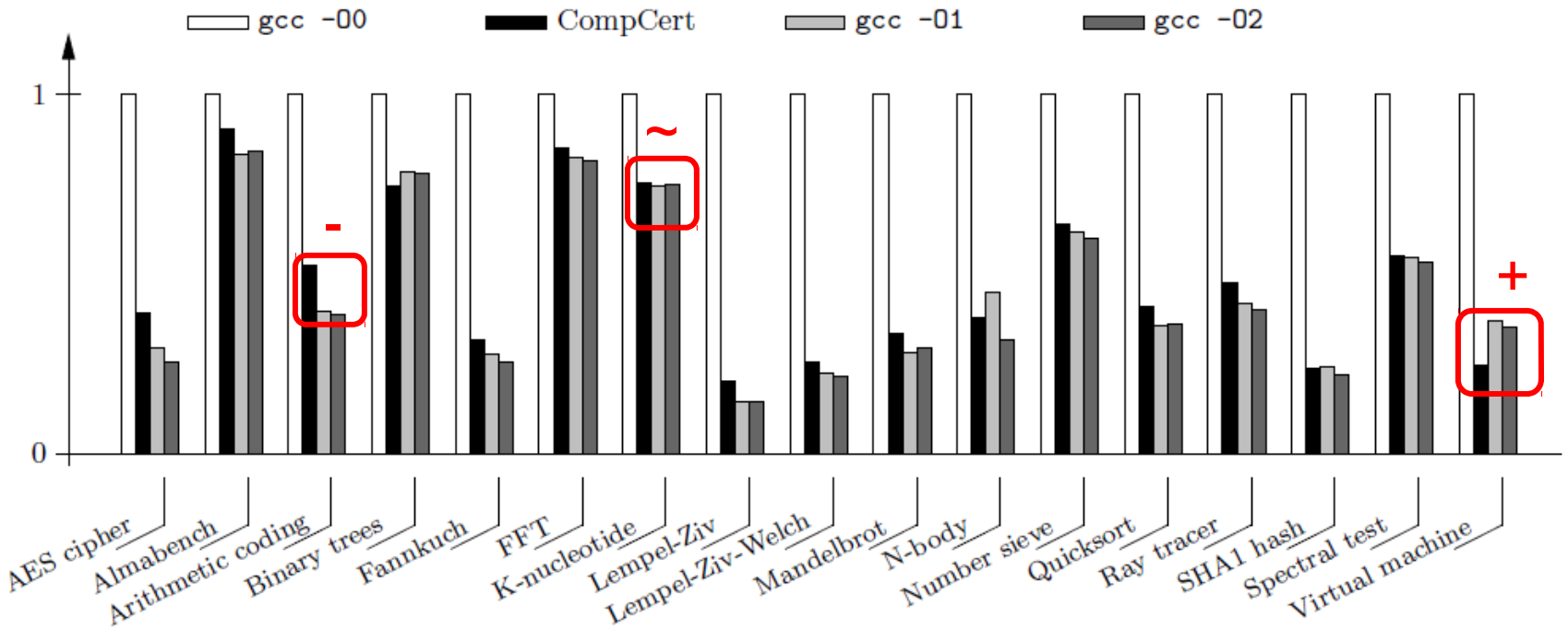
- Sa *correction* a été *entièrement prouvée* dans Coq
  - Correction = code assembleur généré *sémantiquement équivalent* au source initial

- Objectif à terme : compilateur de production pour le logiciel des *A3xx* d'Airbus



CompCert : performances

- *Performances* de CompCert proche de `gcc -O1` ou `-O2`
  - Logiciel certifié ne veut *pas* dire mauvaises performances !



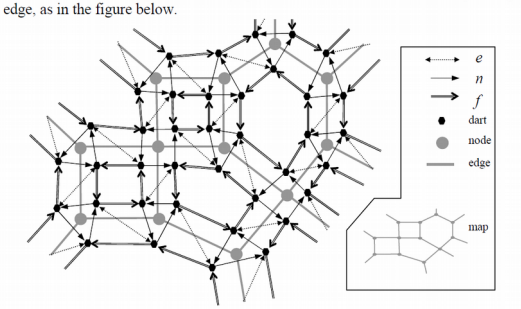
Démonstrateurs interactifs : aperçu

- Outils pour faire un raisonnement proche du *raisonnement mathématique*
  - Coq, Isabelle/HOL, PVS, … (assistants de preuves)

- Permet d'exprimer des *propriétés (très) complexes*
  - Vérifier les règles de preuve de l'Atelier B (Méthode B) dans Coq (travail BiCoq)
  - Démonstration du *théorème des 4 couleurs* avec Coq (Georges Gonthier)

- La logique utilisée est *non décidable*
  - Donc l'*utilisateur* est nécessaire pour guider les preuves

6. Although their name suggests drawing "darts" as arrows on the original map, this leads to horrendously confused figures when one adds arrows for the *e*, *n*, and *f* functions. In the figures below, we therefore always depict darts as points (we use small hexagonal bullets). When we draw an ordinary map together with the corresponding hypermap, we arrange the darts in circles around the corresponding nodes, halfway between adjacent edges. This way, each node appears at the center of the corresponding *n* cycle, each *f* cycle is inset in the corresponding face, and each *e* cycle is the diagonal of an *n*-*f* quadrilateral centered on the corresponding edge, as in the figure below.



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 : *correspondance 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 (cf. CompCert)

Un exemple en Coq

- Une preuve en *Coq* et le programme *extrait*

```
Require Import Omega.

Lemma minus : forall x y : nat, y <= x -> { z | x = y + z }.
Proof.
  induction x as [|x IH].
  { exists 0. omega. }
  intros [|y] ?.
  { exists (S x). omega. }
  destruct (IH y) as [z Hz]. omega. exists z. omega.
Show Proof.
Defined.

Extraction minus.
```

∀x,y:ℕ tq y ≤ x, ∃z:ℕ tq x-y = z

Démo !

```
(** val minus : nat -> nat -> nat **)
let rec minus n y = (* n – y = ? *)
  match n with
  | O -> O  (* 0 – y = 0 (car y ≤ x donc y ≤ 0 donc y = 0) *)
  | S n0 ->  (* 1 + n0 – y = ? *)
    (match y with
     | O -> S n0 (* 1 + n0 – 0 = 1 + n0 *)
     | S y0 -> minus n0 y0) (* 1 + n0 – (1 + y0) = n0 – y0 *)
```

Programmation certifiée : grille de lecture

- Domaines d'application / Problèmes possibles
  - Convient à *tous* types de domaine. Haut niveau de *confiance*
  - Souvent *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
  - Outils *libres*
  - Usage principalement *manuel*
- Expressivité : qu'est-ce que je peux prouver ?
  - Prouver *tous* types de propriétés

Pour conclure

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)
    - Conserver la *structure* de structures de données, *séparer* des entiers ayant un sens différent, faire du *HTML conforme* W3C, …

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 !
  - Cinq approches *principales*
    - Interprétation abstraite, Programmation par contrat, Programmation par raffinement, Model checking et Programmation certifiée
  - Certaines sont *faciles* d'emploi, d'autres *moins*
    - De entièrement *automatiques* à entièrement *manuelles*
  - *Utiles* même si elles ne sont pas complètements employées
    - Même la simple présence d'une spécification formelle est *utile* !
      - IBM CISC information system update : -9% coûts dév. , ÷ 2.5 bugs

Conclusion (2/2)

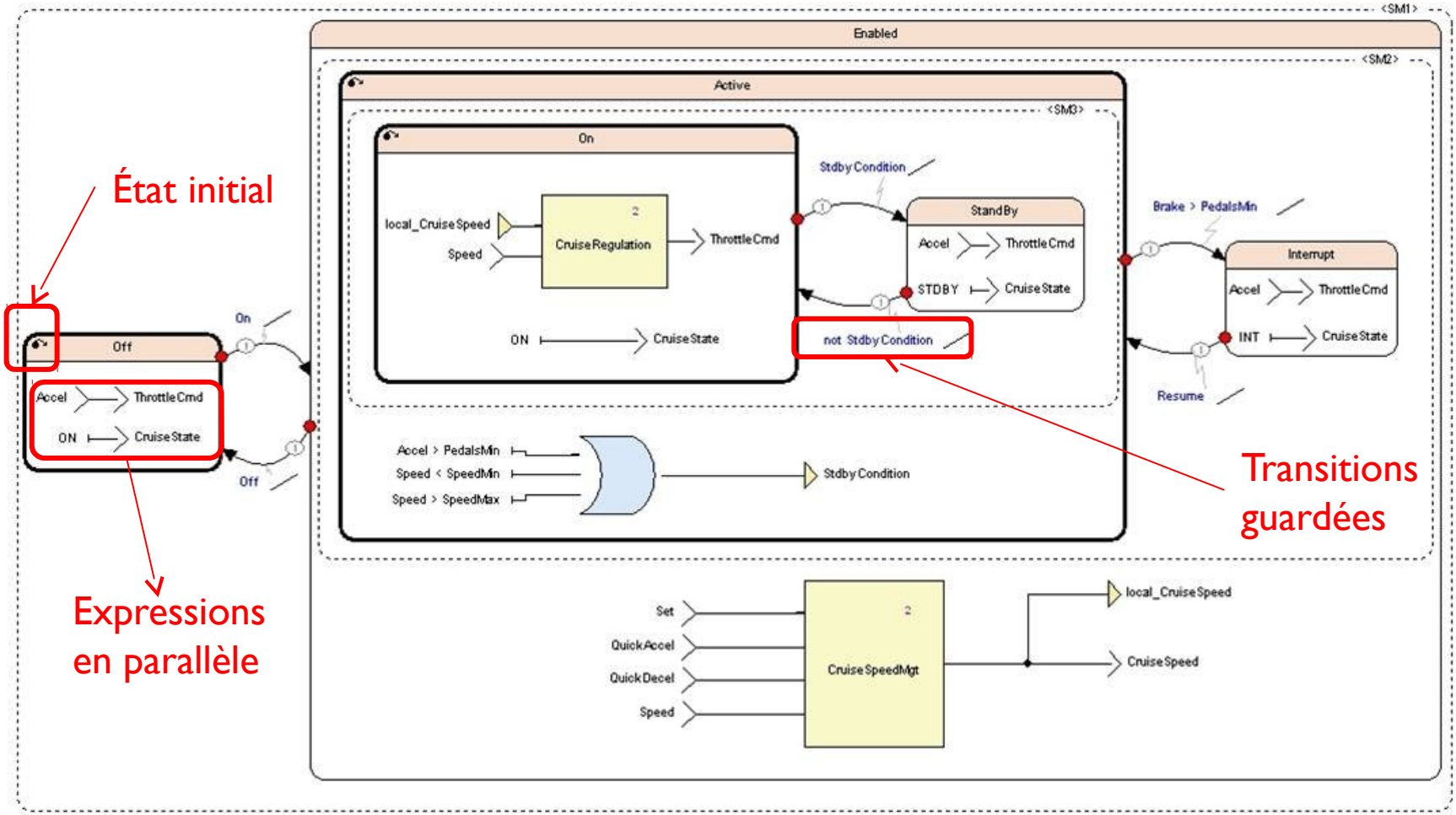
- *Intégrez* les méthodes formelles dans vos développements
  - Comme les tests, un *outil* de plus
    - *(Quasi-)obligatoires* dans certains domaines : ferroviaire, aéronautique, …
    - Prendre des *bonnes* habitudes : les *types* (OCaml, Haskell), les *contrats* !
  - Avoir un *expert* sous la main pour bien les utiliser
    - Projets libres, conseils sur les *listes* de diffusion !
  - Beaucoup d'*outils* sont *disponibles*
    - 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

Model checking : SCADE Suite (A3xx)

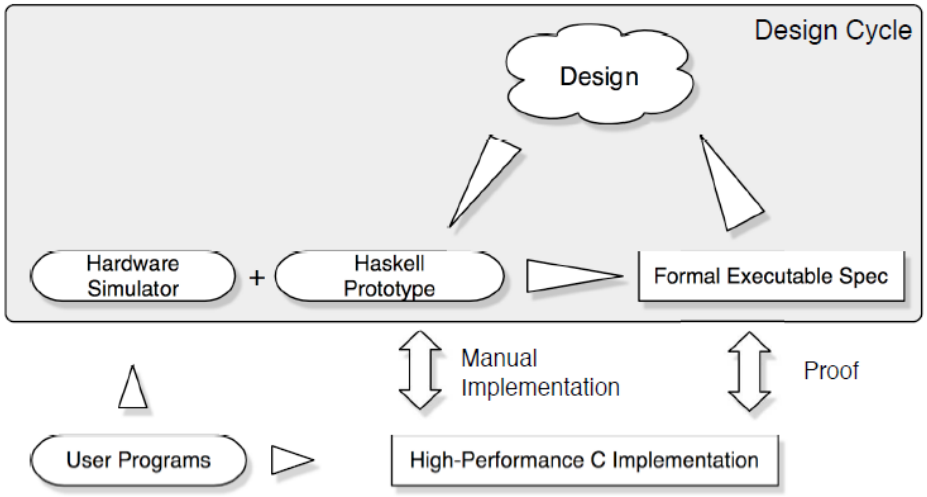


Top Level of the Cruise Control application

Démonstrateurs interactifs : seL4

- seL4 : un micro-noyau de système d'exploitation
- Définition formelle d'un *spécification abstraite*
  - Signification de la correction du noyau
  - Description de ce qu'effectue 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
  - Via l'assistant de preuve Isabelle/HOL



Licence de cette présentation

- Cette présentation est sous licence Art Libre 1.3
  - http://artlibre.org/licence/lal
  - Copyright David MENTRÉ
  - Exception : les images sont propriétés de leurs auteurs