Introduction: Pourquoi formaliser ?

Imene Ben Hafaiedh Marzouk

Spécialité: GLSI

PLAN

Histoire 1 : Intel Pentium FDIV bug

- Bug dans l'unité de division du point-flottant du l'Intel Pentium II découvert au début des années 90.
- Quelques opérations produisent des résultats faux.

Histoire 1 : Intel Pentium FDIV bug

- Bug dans l'unité de division du point-flottant du l'Intel Pentium II découvert au début des années 90.
- Quelques opérations produisent des résultats faux.

- Intel a proposé de remplacer les processeurs défectueux

Histoire 1 : Intel Pentium FDIV bug

- Bug dans l'unité de division du point-flottant du l'Intel Pentium II découvert au début des années 90.
- Quelques opérations produisent des résultats faux.

- Intel a proposé de remplacer les processeurs défectueux
- Cout des pertes estimé à 475 million de dollars

Histoire 1 : Intel Pentium FDIV bug

- Bug dans l'unité de division du point-flottant du l'Intel Pentium II découvert au début des années 90.
- Quelques opérations produisent des résultats faux.

- Intel a proposé de remplacer les processeurs défectueux
- Cout des pertes estimé à 475 million de dollars
- Des Graves conséquences à la réputation d'Intel ! !

Histoire 2 : Ariane 5

- ESA (European Space Agency) Ariane 5 launcher
- L'unique vol du 4 Juin 1996.
- 37 secondes plus tard auto-destruction
- Exception non-interseptée : numerical overflow

Histoire 2 : Ariane 5

- ESA (European Space Agency) Ariane 5 launcher
- L'unique vol du 4 Juin 1996.
- 37 secondes plus tard auto-destruction
- Exception non-interseptée : numerical overflow

- Couteux ! et surtout embarrassant ! ! !

Histoire 3 : Service d'ambulance de Londres

- Système pour dispatcher les ambulances Introduit en 1992.
- Zone 600 $miles^2$, Population 6.8 million, 5000 patients par jour, 2000-2500 appel par jour,

Histoire 3 : Service d'ambulance de Londres

- Système pour dispatcher les ambulances Introduit en 1992.
- Zone 600 $miles^2$, Population 6.8 million, 5000 patients par jour, 2000-2500 appel par jour,

- Position des véhicules mal enregistrée

Histoire 3 : Service d'ambulance de Londres

- Système pour dispatcher les ambulances Introduit en 1992.
- Zone 600 $miles^2$, Population 6.8 million, 5000 patients par jour, 2000-2500 appel par jour,

- Position des véhicules mal enregistrée
- Plusieurs véhicules envoyées à la même destination

Histoire 3 : Service d'ambulance de Londres

- Système pour dispatcher les ambulances Introduit en 1992.
- Zone 600 $miles^2$, Population 6.8 million, 5000 patients par jour, 2000-2500 appel par jour,

- Position des véhicules mal enregistrée
- Plusieurs véhicules envoyées à la même destination
- 20-30 estimés morts à cause de ces erreurs ! ! !

Histoire 4 : Toyota Prius

- Première véhicule **hybride** produite en masse.
- un "glitch" (comportement non-anticipé) trouvé dans le système antiblocage de freinage.
- Plusieurs incidents et plaintes

Histoire 4 : Toyota Prius

- Première véhicule **hybride** produite en masse.
- un "glitch" (comportement non-anticipé) trouvé dans le système antiblocage de freinage.
- Plusieurs incidents et plaintes

- Un Total 185,000 voitures retirées

Histoire 4 : Toyota Prius

- Première véhicule **hybride** produite en masse.
- un "glitch" (comportement non-anticipé) trouvé dans le système antiblocage de freinage.
- Plusieurs incidents et plaintes

- Un Total 185,000 voitures retirées
- Très mauvaise publicité ! ! !

Qu'est ce que ces histoires ont en commun ? ? ?

Qu'est ce que ces histoires ont en commun ? ? ?

- Systèmes informatiques programmables
- Des systèmes et des réseaux +ou- conventionnels
- Des logiciels embarqués sur des appareils

Qu'est ce que ces histoires ont en commun ? ? ?

- Systèmes informatiques programmables
- Des systèmes et des réseaux +ou- conventionnels
- Des logiciels embarqués sur des appareils

- Les erreurs de programmation sont une cause directe de l'échec
- Les logiciels sont critiques pour la sécurité, le business et la réputation des entreprises

Qu'est ce que ces histoires ont en commun ? ? ?

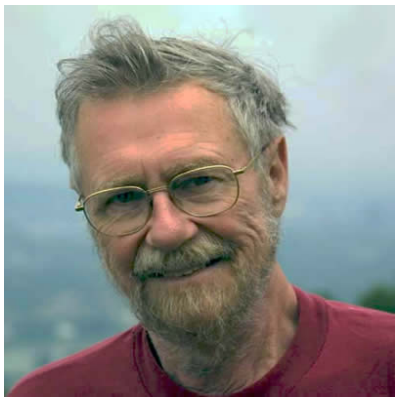
- Systèmes informatiques programmables
- Des systèmes et des réseaux +ou- conventionnels
- Des logiciels embarqués sur des appareils

- Les erreurs de programmation sont une cause directe de l'échec
- Les logiciels sont critiques pour la sécurité, le business et la réputation des entreprises

Ces erreurs peuvent bien être évités ! ! !

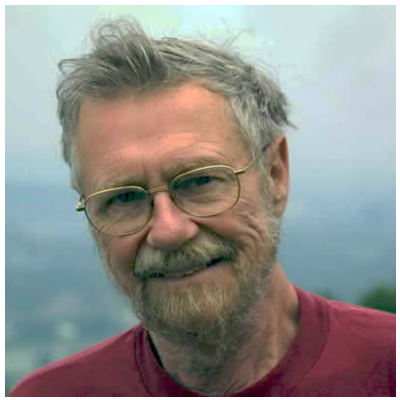
On a besoin de **Vérifier** nos systèmes !

" *Testing can only show the presence of errors, not their absence.* " – ***Edsger Dijkstra (1920-2002)***



On a besoin de **Vérifier** nos systèmes !

" *Testing can only show the presence of errors, not their absence.* " – ***Edsger Dijkstra (1920-2002)***



Pour détecter les erreurs on doit considérer ***toutes les exécutions possibles*** *de notre système même celles impossibles mécaniquement.*

Lois de Murphy relatives aux systèmes :

1. If something can go wrong, it will go wrong.
2. Damage to an object is proportional to its value.
3. Any software bug will tend to maximize the damage.
4. If a system is designed to be tolerant to a set of faults, there will always exist an idiot so skilled to cause a nontolerated fault.
   $\Longrightarrow$ Dummies are always more skilled than measures taken to keep them from harm.
5. If a system stops working, it will do it at the worst possible time.
6. Sooner or later, the worst possible combination of circumstances will happen.
   $\Longrightarrow$ A system must always be designed to resist the worst possible combination of circumstances.

Besoin de Vérifier = Méthodes formelles ?

- Principe des méthodes formelles :

Besoin de Vérifier = Méthodes formelles ?

- Principe des méthodes formelles :

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

Besoin de Vérifier = Méthodes formelles ?

- Principe des méthodes formelles :

  - Utiliser les **mathématiques** pour concevoir et si possible réaliser des systèmes informatiques
  - Spécification formelle / Vérification formelle / Synthèse formelle

Besoin de Vérifier = 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 ambigüité des mathématiques !

Objectif global : améliorer la confiance !

Approche générale

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

Approche générale

- En plusieurs étapes
  1. Spécifier le logiciel en utilisant les mathématiques
  2. Vérifier certaines propriétés sur cette spécification
  3. Corriger la spécification si besoin
  4. (parfois) Raffiner ou dériver de la spécification un logiciel concret
  5. (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, Démonstrateurs de théorèmes automatiques ou interactifs, ...

Les avantages des méthodes formelles

- Re-formuler la spécification en utilisant les mathématiques force à être **précis**
- Un moyen d'avoir des spécifications claires
- 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

Les avantages des méthodes formelles

- Re-formuler la spécification en utilisant les mathématiques force à être **précis**
- Un moyen d'avoir des spécifications claires
- 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

- La **maturité** des outils et leurs usages s'est beaucoup amélioré ces dernières années (mais certains sont toujours très complexes)
- La plupart des outils sont disponibles gratuitement et/ou librement :

`http://gulliver.eu.org/wiki/FreeSoftwareForFormalVerification`

Comment appliquer les MFs ?

Quel est votre problème ?

- Qu'est-ce que vous voulez garantir ?
- Trouver une méthode formelle qui corresponde à :
  - votre domaine d'application
  - vos problèmes
  - vos contraintes de temps et de coûts

Comment appliquer les MFs ?

Quel est votre problème ?

- Qu'est-ce que vous voulez garantir ?
- Trouver une méthode formelle qui corresponde à :
  - votre domaine d'application
  - vos problèmes
  - vos contraintes de temps et de coûts

Comment les mettre en oeuvre ?

- Les **intégrer** à votre **cycle de développement**
- Prendre les conseils d'un **expert** dans la méthode choisie

Grille de choix d'un outil d'une MF ?

- 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
  - A 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 ?

Exemples de Méthodes Formelles

- **Analyse Abstraite**

Exemples de Méthodes Formelles

- **Analyse Abstraite**

- **Model checking**

Exemples de Méthodes Formelles

- Analyse Abstraite
- Model checking
- les Méthodes ensemblistes

Analyse abstraite

- Basée sur l'interprétation abstraite
- Un type abstrait décrit une structure de données en masquant sa réalisation.

l'interprétation abstraite est une théorie d'approximation correcte de la **sémantique** (des modèles) des systèmes informatiques

- **Approximation** : observation du comportement d'un système informatique à un certain niveau d'abstraction, en ignorant les détails sans importance
- **Correcte** : l'approximation ne peut pas mener à des conclusions erronées

Model-checking

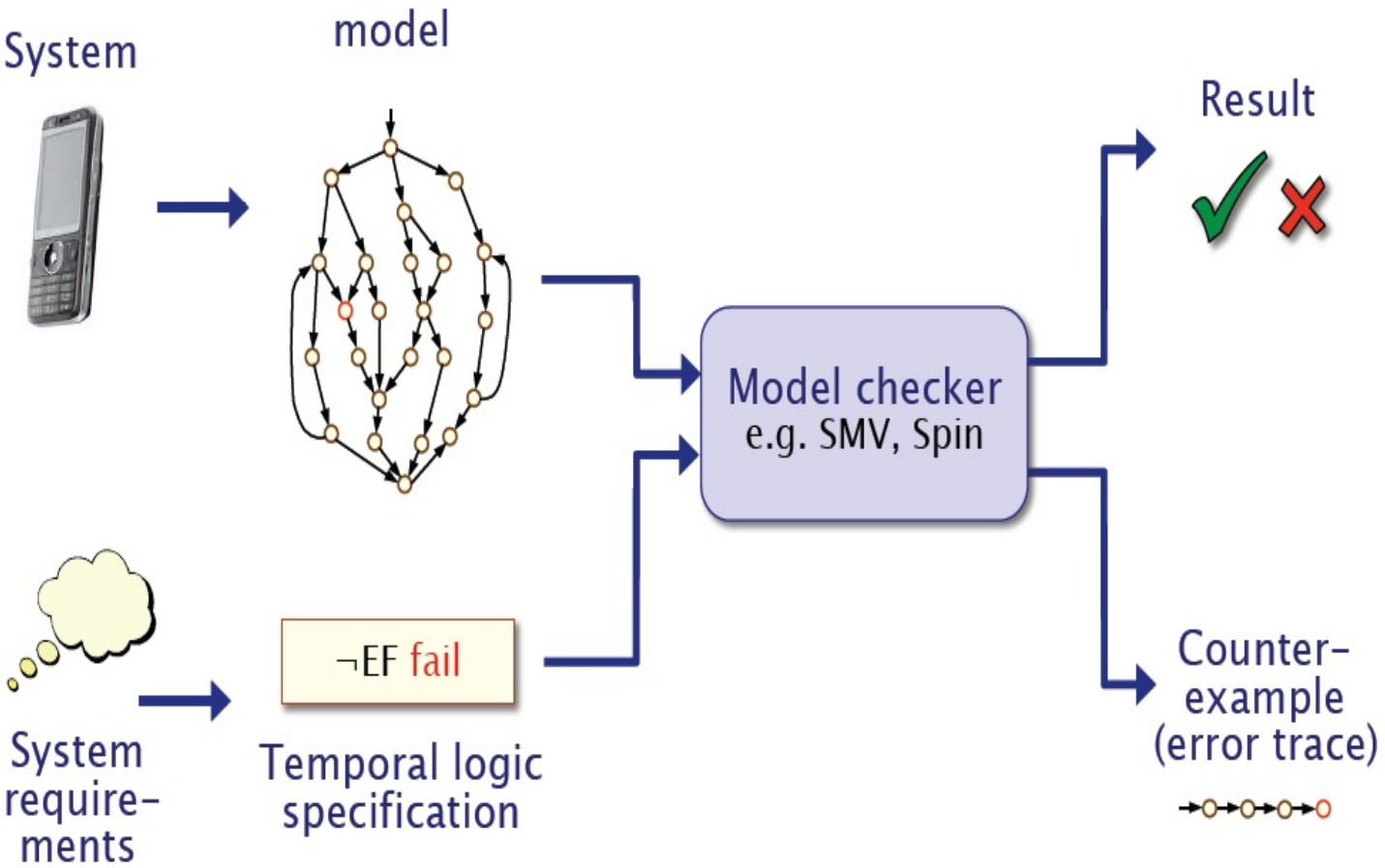
Définition

Méthode **algorithmique** permettant de vérifier **formellement** qu'un **système à états fini** satisfait une **propriété logique**

Nécessite :

1. un **Modèle** M d'un système matériel ou logiciel
2. une description de la propriété φ à vérifier sur ce modèle
3. un algorithme vérifiant automatiquement si le modèle satisfait la formule : $M \models \varphi$

Exemple de MF : Model-checking



Avantages Vs Limitations

Avantages

- (Idéalement) Complètement automatique
- un contre-exemple est retourné quand la propriété n'est pas vérifiée

Avantages Vs Limitations

Avantages

- (Idéalement) Complètement automatique
- un contre-exemple est retourné quand la propriété n'est pas vérifiée

Limitations

- *En théorique :* le système de transitions doit être fini ; grossièrement, le programme ne doit manipuler que des variables à domaine fini.
  C'est souvent le cas en pratique, mais pas toujours.

Méthodes ensemblistes :

Basées sur la théorie axiomatique des **ensembles** e.g. VDM, Z, B

Méthodes ensemblistes :

Basées sur la théorie axiomatique des **ensembles** e.g. VDM, Z, B

La méthode B :

- Construction de logiciels corrects par construction
- Proposée par Jean-Raymond Abrial
- Raffiner une spécification abstraite : En utilisant la logique de Hoare

Méthodes ensemblistes :

Basées sur la théorie axiomatique des **ensembles** e.g. VDM, Z, B

La méthode B :

- Construction de logiciels corrects par construction
- Proposée par Jean-Raymond Abrial
- Raffiner une spécification abstraite : En utilisant la logique de Hoare

La 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

La 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
- Aucun bugs trouvés après les preuves, des tests fonctionnels, 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)

Conclusion

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 (Cette présentation n'est pas exhaustive) !
  - Approches principales : Interprétation abstraite, Model checking, Méthode B
  - Certaines sont faciles, d'autres plus difficiles
  - D'entièrement automatiques à entièrement manuelles
  - Utiles mêmes si elles ne sont pas complètement employées
  - Même une seule spécification formelle est utile !