Preuves et Constructions Formelles

Guilhem Jaber

guilhem.jaber@univ-nantes.fr
http://guilhem.jaber.fr

Comment raisonner mathématiquement sur un système informatique

- $\rightsquigarrow$ Programme, Protocole de communication, Système distribué, Microprocesseur, ...

pour s'assurer de son bon fonctionnement.

Syllabus

Semaine 37	CM	G. Jaber
Semaine 38	TD1	G. Jaber
Semaine 40	TD2	G. Jaber
Semaine 41	TP1	G. Jaber
Semaine 42	TP2	G. Jaber
Semaine 43	TP3	G. Jaber
Semaine 45	CM & TD1	C. Attiogbe
Semaine 46	TD2	C. Attiogbe
Semaine 47	TP1	C. Attiogbe
Semaine 48	TD2	C. Attiogbe
Semaine 49	TP2	C. Attiogbe
Semaine 50	TP3	C. Attiogbe

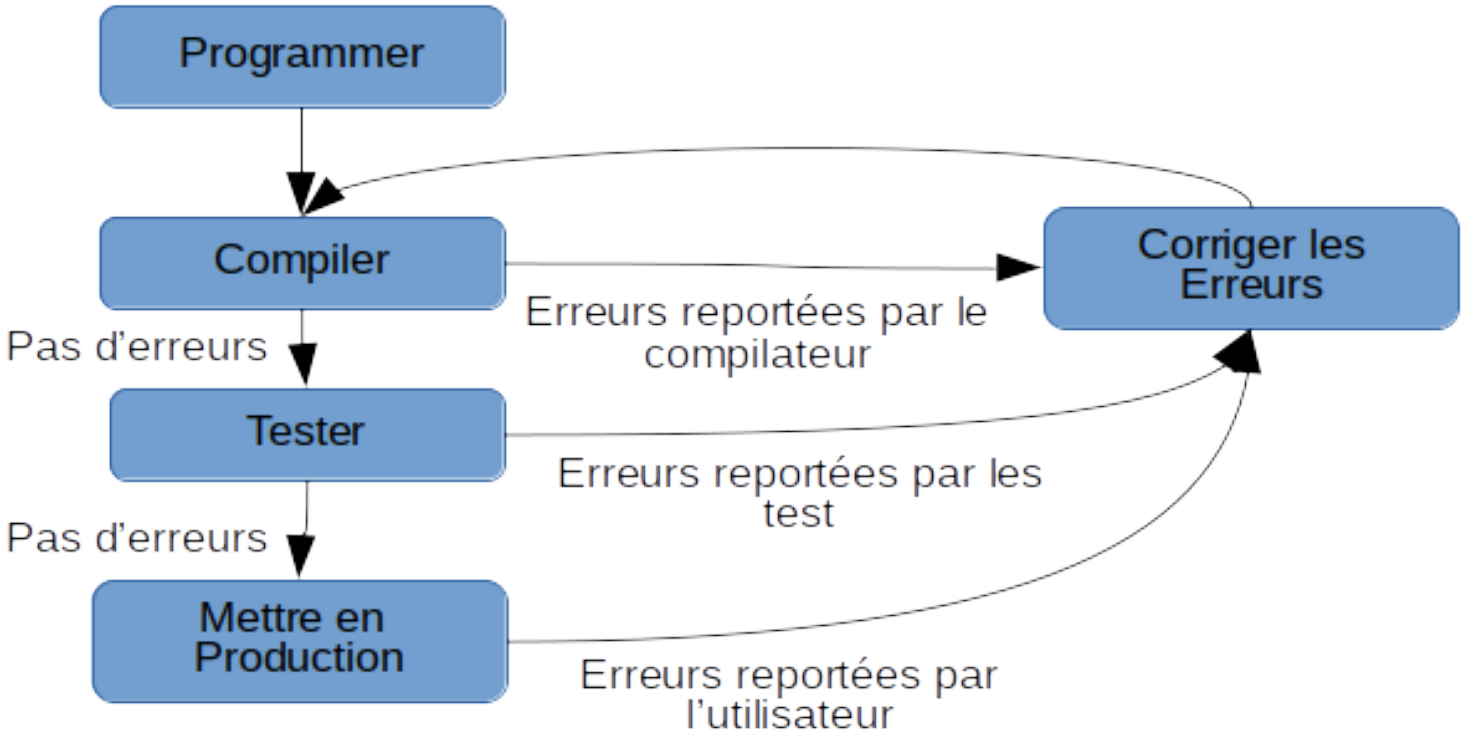
- Evaluation: 2 controles continus, 1 TP noté, 1 projet
- Bientôt une page Madoc !

Concevez-vous souvent un programme non trivial sans aucun bug du premier coup ?

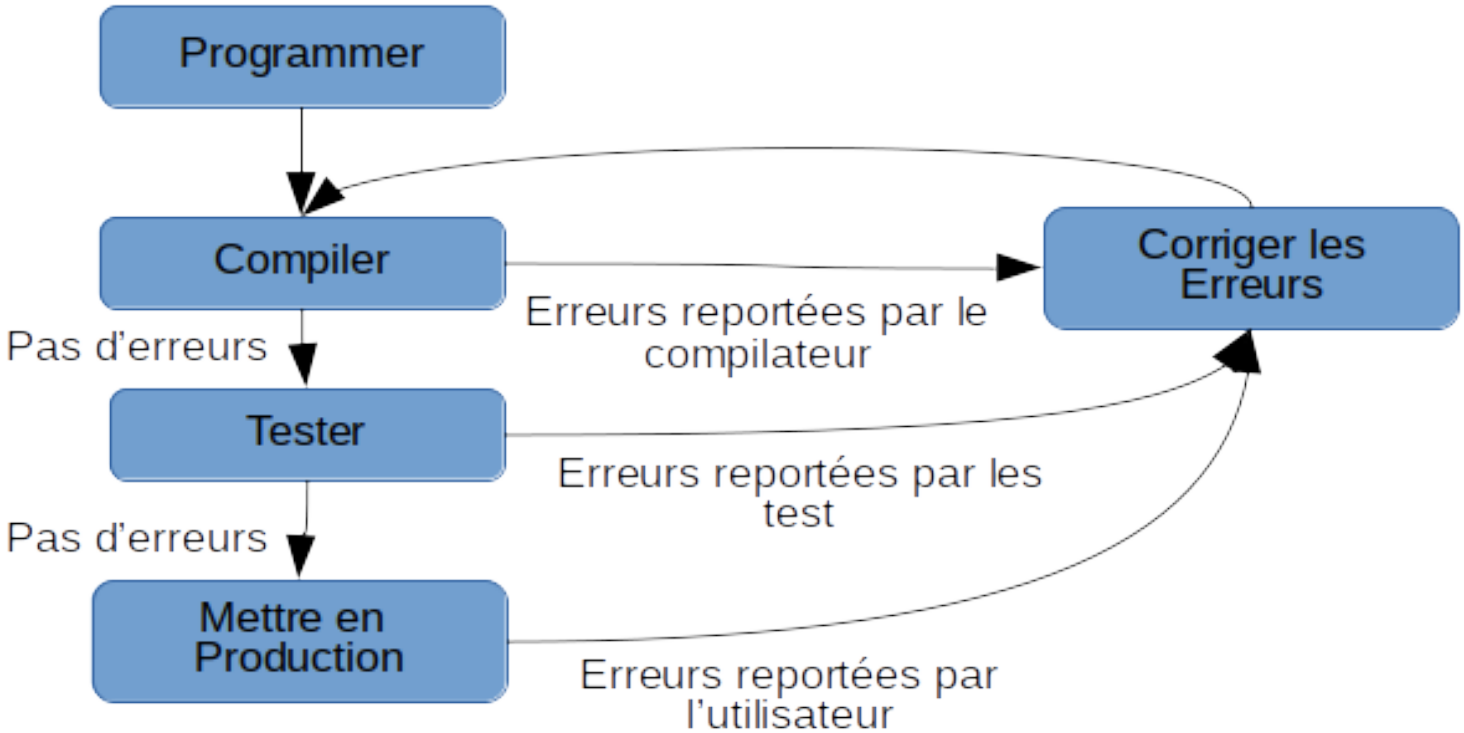
Concevez-vous souvent un programme non trivial sans aucun bug du premier coup ?

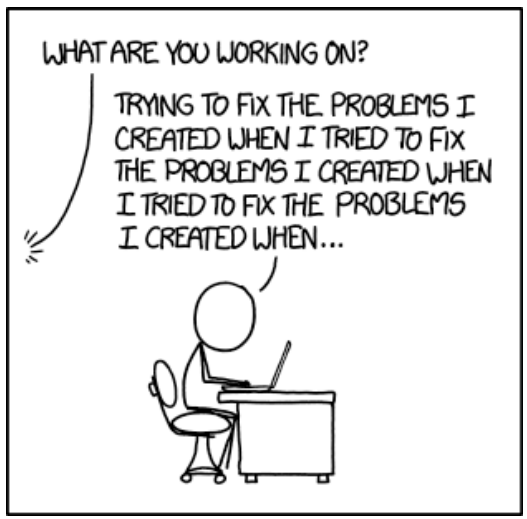
Non ? C'est normal !

Workflow typique du programmeur



Workflow typique du programmeur





Décrivez-moi en quelques mots:

- Un exemple de bug que vous avez eu dans un de vos projets;
- comment vous l'avez trouvé;
- comment vous l'avez corrigé.

Conséquences catastrophiques de la présence de bugs en production.

Ariane 5

- Lors de son lancement initial le 4 juin 1996, Ariane 5 explose en vol après 36,7 secondes.
- L'explosion est due à un bug logiciel.
- Coût: Plusieurs centaines de millions d'euros !

Anatomie du bug logiciel d'Ariane 5

- Due à un dépassement d'entier (integer overflow) à la suite d'une conversion d'un flottant vers un entier.
- Ce dépassement entraîna le déclenchement d'une exception non rattrapée.
- Cette exception entraîna le crash du système de référence inertiel.
- Ce crash fut mal interprète par le système central qui aboutit à un calcul de trajectoire erroné.
- Le code où se situait le bug venait d'Ariane 4.
- Les valeurs flottantes traitées sur Ariane 5 étaient bien plus élevés que sur Ariane 4, d'où le bug non détecté auparavant.

Difficulté de trouver les bugs dans un programme.

Recherche Dichotomique en Java

```
1:  public static int binarySearch(int[] a, int key) {
2:      int low = 0;
3:      int high = a.length - 1;
4:      while (low <= high) {
5:          int mid = (low + high) / 2;
6:          int midVal = a[mid];
7:          if (midVal < key) low = mid + 1
9:          else if (midVal > key) high = mid - 1;
11:           else return mid; // key found
12:     }
13:     return -(low + 1);  // key not found.
14: }
```

Recherche Dichotomique en Java

```
1: public static int binarySearch(int[] a, int key) {
2:     int low = 0;
3:     int high = a.length - 1;
4:     while (low <= high) {
5:         int mid = (low + high) / 2;
6:         int midVal = a[mid];
7:         if (midVal < key) low = mid + 1
9:         else if (midVal > key) high = mid - 1;
11:          else return mid; // key found
12:     }
13:     return -(low + 1);  // key not found.
14: }
```

Erreur à la ligne 5:

- `(low + high) / 2` peut produire un débordement d'entier
- il produit alors un accès fautif dans le tableau à la ligne 6.

Ce bug de la bibliothèque standard Java a mis 9 ans à être détecté !

https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

Meltdown & Spectre

- Failles de sécurité concernant de nombreux microprocesseurs (Intel, AMD, ARM, . . .)
- Contournement de protections d'accès de zones mémoires protégées.

- Situation de compétition (race condition):
  - ⇝ entre la lecture d'une adresse mémoire et la vérification des privilèges permettant de lire à cette adresse;
  - ⇝ due à l'exécution spéculative effectuée par les microprocesseurs pour exécuter des instructions en avance dans le pipeline
  - ⇝ via la prédiction de branchement.
- En cas de mauvaise prédiction, les effets directs de l'exécution sont annulés.
- Cependant des effets indirects peuvent subsister.
  - ⇝ Chargement de la valeur depuis la RAM vers la mémoire cache
  - ⇝ Qui peut être détectée par une attaque par canaux cachés (mesure du temps d'exécution des instructions).

Mais... qu'est-ce-qu'un bug ?

Mais... qu'est-ce-qu'un bug ?

- ⇝ Crash du système;
- ⇝ Système qui ne termine pas;
- ⇝ Faille de sécurité;
- ⇝ **Programme qui ne se comporte pas de la manière prévue.**

Mais... qu'est-ce-qu'un bug ?

- ⇝ Crash du système;
- ⇝ Système qui ne termine pas;
- ⇝ Faille de sécurité;
- ⇝ **Programme qui ne se comporte pas de la manière prévue.**

Comment décrire le comportement attendu d'un système informatique ?

Spécifier le comportement du système

- Spécification évidente: Le système ne "plante pas" (pas de segfault, d'exceptions non rattrapées, ...).
- Raisonner sur le résultat produit par le système: Specification fonctionnelle
  - $\rightsquigarrow$ nécessite un langage précis et non-ambigu pour les exprimer.
- Le système ne divulgue pas d'informations confidentielles.
  - $\rightsquigarrow$ Divulgation indirecte: canaux cachés (temps d'exécution, consommation mémoire, ...).
  - $\rightsquigarrow$ Spécification complexe, domaine de recherche actif !

Un exemple de spécification

- En entrée: Un tableau d'entier A de taille n.
- En sortie: un tableau d'entier B de taille n tel que:
  - pour tout $i \in \{0, \ldots, n-2\}, B[i] \leq B[i+1]$;
  - il existe une permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ tel que $B[i] = A[\sigma(i)]$.

Quel programme vérifie cette spécification ?

Un exemple de spécification

- En entrée: Un tableau d'entier A de taille n.
- En sortie: un tableau d'entier B de taille n tel que:
  - pour tout $i \in \{0, \ldots, n-2\}, B[i] \leq B[i+1]$;
  - il existe une permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ tel que $B[i] = A[\sigma(i)]$.

Quel programme vérifie cette spécification ?

Un programme qui **trie** un tableau d'entier par ordre croissant.

Que signifie qu'un programme vérifie sa spécification ?

Toutes les exécutions du programme ont un comportement satisfaisant la spécification.

Que signifie qu'un programme vérifie sa spécification ?

Toutes les exécutions du programme ont un comportement satisfaisant la spécification.

$\rightsquigarrow$ Vérification dynamique: tester le programme.

Que signifie qu'un programme vérifie sa spécification ?

Toutes les exécutions du programme ont un comportement satisfaisant la spécification.

$\rightsquigarrow$ Vérification dynamique: tester le programme.

Problème: Impossible de vérifier toutes les exécutions possibles.

"Testing shows the presence, not the absence of bugs." E. W. Dijkstra

Que signifie qu'un programme vérifie sa spécification ?

Toutes les exécutions du programme ont un comportement satisfaisant la spécification.

$\rightsquigarrow$ Vérification dynamique: tester le programme.

Problème: Impossible de vérifier toutes les exécutions possibles.

"Testing shows the presence, not the absence of bugs." E. W. Dijkstra

Solution: raisonner mathématiquement pour prouver que toutes les exécutions sont correctes.

Comment raisonner sur les exécutions d'un programme ?

Comment raisonner sur les exécutions d'un programme ?

- Raisonner sur l'exécution du code binaire issu de la compilation du programme.
  - $\rightsquigarrow$ Infaisable: trop complexe, dépendant du processeur sur lequel l'exécution a lieu, du compilateur et du système d'exploitation utilisé.

Comment raisonner sur les exécutions d'un programme ?

- Raisonner sur l'exécution du code binaire issu de la compilation du programme.
  - $\rightsquigarrow$ Infaisable: trop complexe, dépendant du processeur sur lequel l'exécution a lieu, du compilateur et du système d'exploitation utilisé.

- Raisonner sur le code source du programme.
  - $\rightsquigarrow$ Nécessite de donner une **sémantique** au programme.

Sémantique des langages de programmation

- **Sémantique dénotationnelle**:
  Voir un programme comme une fonction mathématique.
  - $\rightsquigarrow$ Calcul de points fixes pour représenter les boucles et la récursion.

Sémantique des langages de programmation

- **Sémantique dénotationnelle**:
  Voir un programme comme une fonction mathématique.
  - $\rightsquigarrow$ Calcul de points fixes pour représenter les boucles et la récursion.

- **Sémantique opérationnelle**:
  représentation mathématique de l'exécution d'un programme.
  - $\rightsquigarrow$ Sémantique opérationnelle structurelle (à petit pas): description de chaque étape intermédiaire de l'exécution du programme.
  - $\rightsquigarrow$ Sémantique opérationnelle naturelle (à grand pas): description du résultat final de l'exécution du programme.
  - $\rightsquigarrow$ Système de transitions dont les états sont de la forme (i, s) avec i une instruction et s un état mémoire.

Décrire l'évolution de la mémoire au cour de l'exécution du programme suivant:

```
int n = 3;
int base = 2;
int pos = 1;
int result = 0;
while (n > 0) {
   result = result+ n*pos;
   pos = pos*base;
   n = n-1;
}
```

Difficulté de définir la sémantique opérationnel d'un vrai langage de programmation:

```
int foo() {
  printf(``foo'');
  return 1;
}

int bar() {
  printf(``bar'');
  return 3;
}

void main() {
  int n = foo() + bar();
  printf(``\%d'',n);
}
```

Difficulté de définir la sémantique opérationnel d'un vrai langage de programmation:

```
int foo() {
  printf(``foo'');
  return 1;
}

int bar() {
  printf(``bar'');
  return 3;
}

void main() {
  int n = foo() + bar();
  printf(``\%d'',n);
}
```

- Comportement non spécifié en C.
- Choix du compilateur.

Comment s'assurer que le système **satisfait sa spécification** ?

Deux approches:

- Système correct par construction.
  - ⇝ Synthèse d'un système à partir de sa spécification.
  - ⇝ Construction du système par raffinement successifs de sa spécification.
  - ⇝ Méthode B.
  - ⇝ **Deuxième partie de ce cours !**
- **Raisonnement a posteriori.**

Utiliser le raisonnement logique pour prouver que le système satisfait sa spécification.

Est-il possible de concevoir une méthode **automatique** prenant en entrée une description d'un programme P et une spécification ϕ, et qui détermine si P verifie ϕ ?

Est-il possible de concevoir une méthode **automatique** prenant en entrée une description d'un programme P et une spécification ϕ, et qui détermine si P verifie ϕ ?

Non ! n'importe quelle propriété "non-trivial" spécifiant les exécutions d'un programme est indécidable. (Théorème de Rice).

- méthode automatique $\rightsquigarrow$ Programme informatique.
- Description d'un programme $\rightsquigarrow$ Code source.

Problème de l'arrêt

Il n'existe pas de méthode automatique prenant en entrée une description d'un programme P et une entrée u, et qui détermine si $P(u)$ termine.

Prouvé par Alan Turing en 1936

- Avant l'introduction des ordinateurs !
- Programme $\rightsquigarrow$ Machine de Turing.
- Mémoire non bornée.

Toute tentative de vérification de programmes est-elle alors vouée à l'échec ?

Toute tentative de vérification de programmes est-elle alors vouée à l'échec ?

Non ! Il existe plusieurs possibilités:

- Le langage de programmation restreint les programmes considérés, pour qu'ils se comportent tous correctement.
  - $\rightsquigarrow$ Système de type.
  - $\rightsquigarrow$ Des programmes valides ne sont alors pas acceptés.
- L'utilisateur aide la méthode pour effectuer la vérification.
  - $\rightsquigarrow$ Raisonnement déductif.
- La méthode effectue des approximations, pour se ramener à un problème décidable.
  - $\rightsquigarrow$ Model-checking, analyse statique.
  - $\rightsquigarrow$ Présence de fausses alarmes (faux-positifs).

Interlude logique

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

- Logique propositionnelle: Variables propositionnelles P, Q et connecteurs $\wedge, \vee, \neg, \Rightarrow$
  - $\rightsquigarrow$ $P \wedge Q \Rightarrow P \vee Q$

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

- Logique propositionnelle: Variables propositionnelles P, Q et connecteurs $\wedge, \vee, \neg, \Rightarrow$
  - $\rightsquigarrow$ $P \wedge Q \Rightarrow P \vee Q$
- Logique des prédicats: Ajout de variables x, y et des connecteurs $\exists, \forall$ à la logique propositionnelle.
  - $\rightsquigarrow$ $(\forall x.P(x) \Rightarrow Q(x)) \wedge (\exists y.P(y)) \Rightarrow \exists z.Q(z)$

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

- Logique propositionnelle: Variables propositionnelles P, Q et connecteurs $\wedge, \vee, \neg, \Rightarrow$
  - $\rightsquigarrow$ $P \wedge Q \Rightarrow P \vee Q$
- Logique des prédicats: Ajout de variables x, y et des connecteurs $\exists, \forall$ à la logique propositionnelle.
  - $\rightsquigarrow$ $(\forall x.P(x) \Rightarrow Q(x)) \wedge (\exists y.P(y)) \Rightarrow \exists z.Q(z)$
- Arithmétique: Ajout des symboles $+, *, <, =, 0$ à la logique des prédicats.
  - $\rightsquigarrow$ $\forall x.(x > 0) \Rightarrow \exists y.x = y + 1$

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

- Logique propositionnelle: Variables propositionnelles P, Q et connecteurs $\wedge, \vee, \neg, \Rightarrow$
  - $\rightsquigarrow$ $P \wedge Q \Rightarrow P \vee Q$
- Logique des prédicats: Ajout de variables x, y et des connecteurs $\exists, \forall$ à la logique propositionnelle.
  - $\rightsquigarrow$ $(\forall x.P(x) \Rightarrow Q(x)) \wedge (\exists y.P(y)) \Rightarrow \exists z.Q(z)$
- Arithmétique: Ajout des symboles $+, *, <, =, 0$ à la logique des prédicats.
  - $\rightsquigarrow$ $\forall x.(x > 0) \Rightarrow \exists y.x = y + 1$
- Théorie des ensemble: Ajout des symboles $\in, =$ à la logique des prédicats.

La logique mathématique fournit un langage formel permettant d'écrire des énoncés de manière précise et non-ambigu.

- Logique propositionnelle: Variables propositionnelles P, Q et connecteurs $\wedge, \vee, \neg, \Rightarrow$
  - $\rightsquigarrow$ $P \wedge Q \Rightarrow P \vee Q$
- Logique des prédicats: Ajout de variables x, y et des connecteurs $\exists, \forall$ à la logique propositionnelle.
  - $\rightsquigarrow$ $(\forall x.P(x) \Rightarrow Q(x)) \wedge (\exists y.P(y)) \Rightarrow \exists z.Q(z)$
- Arithmétique: Ajout des symboles $+, *, <, =, 0$ à la logique des prédicats.
  - $\rightsquigarrow$ $\forall x.(x > 0) \Rightarrow \exists y.x = y + 1$
- Théorie des ensemble: Ajout des symboles $\in, =$ à la logique des prédicats.

$\rightsquigarrow$ Syntaxe

Raisonnement Logique

- Règles d'inférences: $\dfrac{A_1 \quad \dots \quad A_n}{C}$
  - $A_1, \dots, A_n$ sont les prémisses de la règle.
  - C est la conclusion.
  - Règle sans premisse: axiome.
- Par exemple: $\dfrac{P \Rightarrow Q \quad P}{Q}$

- Preuve: succession de règles d'inférences qui se terminent par des **axiomes**.
  - Example d'axiome de l'arithmétique: $\forall x, x + 1 \neq 0$.
- Les preuves ont une structure d'arbre.

Sémantique

- **Modèle** d'une formule: valuation des variables telle que la formule soit vraie.
  - Exemple d'un modèle de $(P \wedge Q) \vee \neg Q$: $[P \mapsto \text{faux}, Q \mapsto \text{faux}]$.
- Modèle d'une théorie: modèle dans laquelle les axiomes sont vrais.
- **Satisfiabilité** d'une formule: existence d'un modèle dans laquelle la formule est vraie.
  - $(P \wedge Q) \vee \neg Q$ est satisfiable.
- **Validité** d'une formule: la formule est vraie dans tout modèle.
  - $(P \wedge Q) \vee \neg Q$ n'est pas valide: elle est fausse dans le modèle $[P \mapsto \text{faux}, Q \mapsto \text{vrai}]$

Quelques Résultats

- Si une formule d'une théorie est prouvable, alors elle est valide (c-à-d vraie dans tous modèle de la théorie).
- Il existe un algorithmique permettant de décider si une formule de la logique propositionnelle est satisfiable.
  - Complexité: NP-complet.
  - En pratique: des solveurs très efficaces (solveurs SAT).
- Il ne peut pas exister d'algorithme pour décider si une formule de la logique des prédicats/de l'arithmétique/de la théorie des ensembles est satisfiable.
- Mais il existe des **fragments décidables**
  - Arithmétique de Presburger, Corps réels clos, ...
  - Solveurs SMT (Satisfiabilité Modulo Théorie): z3, Alt-Ergo, CVC, yice, ...

Fin de l'interlude.

Retour aux méthodes pour s'assurer qu'un programme vérifie une spécification:

- Model-checking.
- Analyse statique.
- Systèmes de Types.
- Vérification deductive.

Model-Checking

Cours de M2: Modélisation et Vérification des Systèmes Concurrents

Model-Checking

Vérification automatique qu'un modèle satisfait une spécification.

$$\mathcal{M} \models \phi$$

- Modèle $\mathcal{M}$: Système de transition.
  - $\rightsquigarrow$ **Structure de Kripke**.
- Specification ϕ : Formule de **logique temporelle**.
  - $\rightsquigarrow$ Ajouts de **modalités** à la syntaxe de la logique.
  - $\rightsquigarrow$ Modalité $\Box\phi$: pour tout état futur, ϕ est vrai.
  - $\rightsquigarrow$ Modalité $\Diamond\phi$: il existe état futur dans lequel ϕ est vrai.

Restreint à des problèmes pour lequel $\mathcal{M} \models \phi$ est decidable.

- $\mathcal{M}$ doit en général être fini.

Modèles

Grande variété de modèles possibles:

- Automates.
- Systèmes réactif.
- Systèmes probabiliste.
- Systèmes temps reels.
- Systèmes hybrides (combinaison de comportements continus et discrets).

Nécessite d'avoir un modèle fini du système que l'on veut analyser.

- Ne peut pas s'appliquer directement au code source d'un programme.
- Techniques d'abstraction pour transformer un programme en modèle fini.

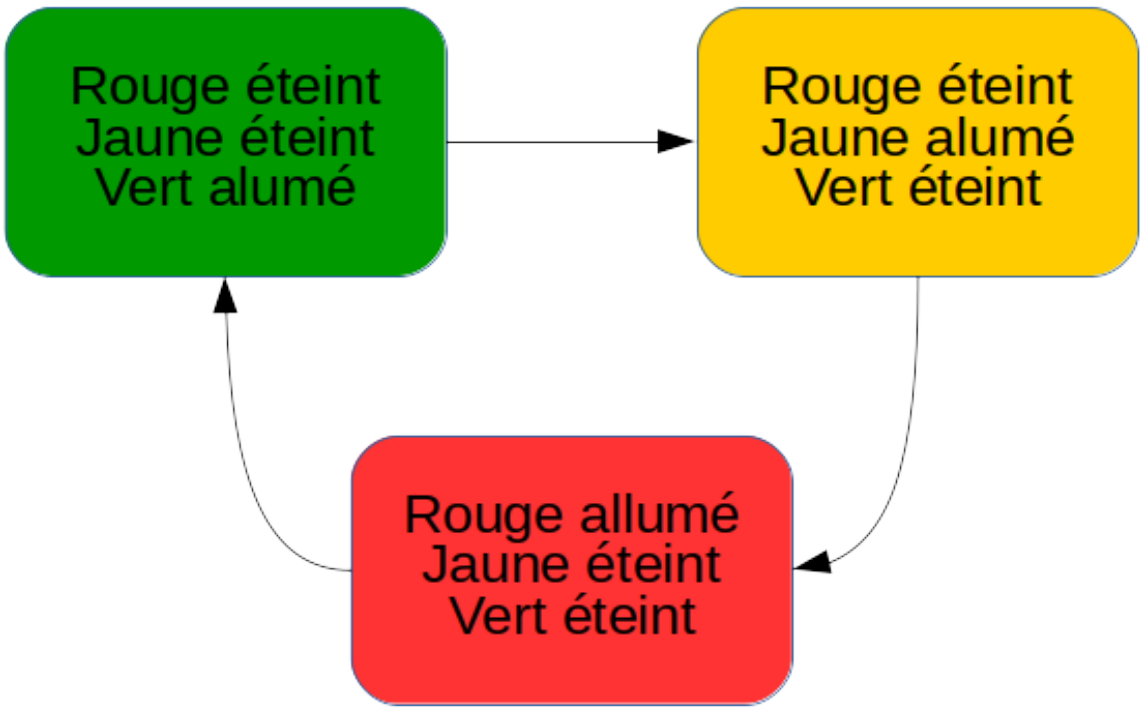
Exemple de spécifications:

- Sûreté (safety): quelque chose de mauvais n'arrive jamais.
  - $\rightsquigarrow$ Le système ne crash pas.

- Vivacité (liveness): une bonne chose finit par arriver.
  - $\rightsquigarrow$ Absence de deadlock: des processus exécutés en parallèle qui s'attendent mutuellement.
  - $\rightsquigarrow$ Absence de famine: un processus en attente qui ne reprend jamais la main.

Un Exemple

Feu de la circulation



- propriété de sûreté: les feus rouge et jaune ne sont jamais allumés en même temps;
- propriété de vivacité: le feu finit toujours par passer au vert.

Problématiques du Model-Checking

- Challenge algorithmique:
  Traiter le problème de l'explosion du nombre d'états ("state-explosion problem").
  - $\rightsquigarrow$ Particulièrement présent dans l'analyse de systèmes concurrents.
  - $\rightsquigarrow$ Utilisation des symétries du modèle.
  - $\rightsquigarrow$ Représentation symbolique: diagrammes de décision binaire.

- Challenge de la modélisation:
  Abstraction pour se ramener à un nombre fini d'états.
  - $\rightsquigarrow$ Raffinement de l'abstraction guidé par les contre-exemples (CEGAR).

Utilisation du Model-Checking

- Hardware: vérification de circuits électroniques représentés dans des langages de description (VHDL, Verilog).
- Analyse de systèmes de processus concurrents (SPIN).
- Analyse de systèmes temps réels (UPPAAL).
- Analyse de programmes (BLAST, CBMC, Microsoft SLAM pour l'analyse de drivers).

Analyse Statique

Analyse Statique

Vérifier automatiquement des propriétés d'un programme à partir de son code source.

En général des propriétés indécidables:

- Analyse de flot de données.
  - $\rightsquigarrow$ Quelle valeur peut prendre une variable à un point donné du code source ?
- Analyse de flot de contrôle
  - $\rightsquigarrow$ Déterminer l'ordre dans lequel les instructions/procédures sont exécutées.
- Détection des accès mémoires fautifs
  - $\rightsquigarrow$ Déreferencement du pointeur NULL, accès aux cases d'un tableau en dehors de ses bornes.

Sûreté de l'analyse statique

L'Analyse statique repose sur des **approximations sûres** du comportement des programmes:

- Préserve la correction (soundness): si l'analyse détermine que le programme vérifie cette propriété, alors c'est le cas !
- Analyse incomplète: si l'analyse détermine que le programme ne vérifie cette propriété, elle peut se tromper
  - $\rightsquigarrow$ Fausses alarmes (false-positives).
- Théorie de l'approximation: **Interprétation abstraite**.
  - $\rightsquigarrow$ Exemple le plus simple: représenter un nombre par son signe.

Utilisation de l'analyse statique

- Compilation:
  - $\rightsquigarrow$ pour permettre des optimisations agressives du code généré.
- Informatique embarquée (Aérospatiale): Analyseur Astrée (basé sur l'interprétation abstraite).
  - $\rightsquigarrow$ Pour s'assurer de l'absence de grandes classes de bugs (segfault, accès à des tableaux en dehors des bornes, ...).
- De nombreux outils industriels non-sûrs (Coverity, PVS-studio, ...)
  - $\rightsquigarrow$ Utiles pour trouver des bugs, mais n'apportent aucune garantie sur l'absence de bugs.

Système de Types

Système de Types (I/II)

- Associe des types aux différentes constructions d'un langage de programmation.
  - $\rightsquigarrow$ $\mathrm{Int}, \mathrm{Bool}, \mathrm{Float}, \mathrm{String}$
  - $\rightsquigarrow$ $T*$: Pointeur vers un élément de type T.
  - $\rightsquigarrow$ $T[]$: Tableau stockant des éléments de type T.

Système de Types (I/II)

- Associe des types aux différentes constructions d'un langage de programmation.
  - $\rightsquigarrow$ $\mathrm{Int}, \mathrm{Bool}, \mathrm{Float}, \mathrm{String}$
  - $\rightsquigarrow$ $T*$: Pointeur vers un élément de type T.
  - $\rightsquigarrow$ $T[]$: Tableau stockant des éléments de type T.
- Type $T \to U$ dans les langages fonctionnels (OCaml, Haskell).

Système de Types (I/II)

- Associe des types aux différentes constructions d'un langage de programmation.
  - $\rightsquigarrow$ $\mathrm{Int}, \mathrm{Bool}, \mathrm{Float}, \mathrm{String}$
  - $\rightsquigarrow$ $T*$: Pointeur vers un élément de type T.
  - $\rightsquigarrow$ $T[]$: Tableau stockant des éléments de type T.
- Type $T \rightarrow U$ dans les langages fonctionnels (OCaml, Haskell).
- Construction de nouveaux types de donnés dans un programme
  - $\rightsquigarrow$ Types énumérés:
    `enum couleur_carte {Coeur, Pique, Trefle, Carreau};`
  - $\rightsquigarrow$ Structure regroupant différente composante:
    `struct carte {couleur_carte couleur; int valeur;}`
  - $\rightsquigarrow$ Types algébriques: `type list = |Nil |Cons of (int * list)`

Système de Types (II/II)

- Typage statique: le compilateur vérifie la cohérence entre le type des différentes parties du programme.
  - `int x = "foo";`
  - `let f x = x + 1 in f true`

Système de Types (II/II)

- Typage statique: le compilateur vérifie la cohérence entre le type des différentes parties du programme.
  - `int x = "foo";`
  - `let f x = x + 1 in f true`
- Typage explicite (C, C++, Java): obligation d'indiquer les informations de type dans le programme.
  - $\rightsquigarrow$ Mais conversion implicite entre type en C: par exemple de char vers int.
- Typage implicite (Haskell, OCaml): le compilateur infère automatiquement les types des composants.

Sûreté du typage

Déduire des propriétés sur l'exécution d'un programme à partir du fait qu'il est bien typé.

- Sûreté mémoire: l'exécution d'un programme bien typé ne génère pas de segfault.
  - ⇝ Nécessite que les types ne soit pas "nullable".
  - ⇝ Pas de valeur NULL qui habite tous les types.
  - ⇝ Nécessite d'initialiser les variables au moment de leur déclaration.
  - ⇝ Utilisation de types options pour indiquer explicitement qu'un type contient une valeur dénuée de sens (None en OCaml, Rust ou Scala).

"Well-typed programs cannot "go wrong"." R. Milner

Des types de plus en plus riches

- Modularité via le système de type: Types abstraits, polymorphisme (generics en Java & Scala).
- Unités de mesure (F#): types pour les unités physiques (kg, m, . . .).
- Types "ownership" (Rust): Contrôler les structures qui peuvent être dupliquées ou modifiées.
  - ⇝ Contrôle fin de l'aliasing.
  - ⇝ Permet d'éviter les data-races en présence de concurrence.
- Types dépendants (Coq, Agda, Idris): Types de donnés indexés par des valeurs.
  - ⇝ Par exemple: type des listes de taille n.
  - ⇝ Permet d'écrire des spécifications très précises pour les programmes.
  - ⇝ Nécessite de typer explicitement les programmes via des annotations.
  - ⇝ Interaction avec les effets de bord problématique: programmation fonctionnelle pure.

Les langages typés permettent d'éviter de nombreux bugs.

De plus en plus utilisés:

- Surcouche à Javascript: Typescript (Microsoft), Flow (Facebook).
- Succès de Rust et de Scala.
- Renouveau des langages fonctionnels typés: Haskell, OCaml.

Vérification Déductive

Logique de Hoare

- Raisonnement sur la structure du code source du programme.
- Via des règles d'inférence (raisonnement déductif).
- Supposition sur les états mémoires au départ de l'exécution: **pre-condition**.
- Attente sur les états mémoires à la fin de l'exécution: **post-condition**.

$$\{P\}c\{Q\}$$

Invariant de boucle

$$\frac{\{P \wedge b\}c\{P\}}{\{P\}\texttt{while } b \texttt{ do } c\{P \wedge \neg b\}}$$

P est l'invariant de boucle. Il est vérifié:

- avant la boucle;
- à chaque itération de la boucle;
- à la fin de l'exécution de la boucle.

Un exemple

Trouver un invariant de boucle pour le programme suivant:

```
int max(int tab[], int taille) {
  int result = tab[0];
  int i = 1;
  while (i < taille) {
      if (result < tab[i]) { result = tab[i]; }
      i = i+1;
  }
  return m;
```

Un exemple

Trouver un invariant de boucle pour le programme suivant:

```
int max(int tab[], int taille) {
  int result = tab[0];
  int i = 1;
  while (i < taille) {
      if (result < tab[i]) { result = tab[i]; }
      i = i+1;
  }
  return m;
```

result est égal à la valeur maximale dans $\mathrm{tab}[0 \ldots i-1]$

Programmation par contrats

Problème: les invariants de boucle ne peuvent pas en général être trouvés automatiquement.

Programmation par contrats

Problème: les invariants de boucle ne peuvent pas en général être trouvés automatiquement.

Solution:

- Laisser le programmeur **annoter** le code source avec ces invariants
- S'assurer ensuite qu'ils sont bien vérifiés:
  - Dynamiquement: à l'exécution du programme
    - $\rightsquigarrow$ Test de la spécification.
  - Statiquement: via du raisonnement logique.
    - $\rightsquigarrow$ Utilisation de solveurs SMT.
    - $\rightsquigarrow$ Mais pas toujours automatisable.
- Outils: Eiffel, Frama-C, F*.

Comment s'assurer que le raisonnement effectué sur un programme est correct ?

$\rightsquigarrow$ Si l'on peut faire des erreurs lorsque l'on programme, on peut aussi en faire lorsque l'on prouve qu'un programme est correct !

Solution: Assistants de preuves permettant de formaliser puis de vérifier qu'une preuve est correcte.

Assistants de Preuves

Logiciel permettant d'aider l'utilisateur à produire une preuve formelle:

- Fourni un cadre pour écrire des énoncés logiques et les prouver.
- Permet d'automatiser une partie du raisonnement.
- Vérifie une fois la preuve terminée qu'elle est correcte.

De nombreux outils existants: Coq, Agda, Lean, PVS, HOL, Isabelle, ACL2, ...

Correspondance de Curry-Howard

Une preuve mathématique peut être vue comme un programme informatique.

Logique	Programme
Formule logique	Type
Preuve	Programme fonctionnel pur
Élimination des coupures	Exécution du programme

- Correspondance basée sur le λ-calcul typé:
$$\frac{\Gamma \vdash M : P \Rightarrow Q \quad \Gamma \vdash N : P}{\Gamma \vdash MN : Q} \qquad \frac{\Gamma, x : P \vdash M : Q}{\Gamma \vdash \lambda X.M : P \Rightarrow Q}$$
- Nécessite un système de type très riche (types dépendants, polymorphisme)
- Raisonnement logique **intuitionniste**: $P \vee \neg P$ n'est pas prouvable.
- Cadre à de nombreux assistants de preuves: Coq, Agda, Lean.

Utilisation des Assistants de Preuves

- Reste encore restreint à un public expert.
- Compilateur certifié en Coq: Compcert (performance proche de gcc).
- Système d'exploitation certifié en Isabelle: micro-noyau seL4.
- Formalisation de preuves mathématiques: Théorème des quatre couleurs (graphes planaires), Théorème de Feit-Thomson (théorie des groupes), Conjecture de Kepler (empilement de sphères).

Un premier bilan

- La vérification de systèmes est bien implantées dans certains domaines:
  - $\rightsquigarrow$ Hardware, Protocoles de communications: Model-checking.
  - $\rightsquigarrow$ Informatique embarquées: Analyse statique (Astrée, ...).
- Les outils d'analyse statique de code se démocratise:
  - $\rightsquigarrow$ Infer chez Facebook.
  - $\rightsquigarrow$ Dafny, CodeContract & F* chez Microsoft.
- Le typage prend une place de plus en plus importante pour s'assurer du bon fonctionnement des programmes.

Les concepts clés

- Spécification d'un système informatique.
- Sémantique d'un langage de programmation.
- Logique: Syntaxe, Modèle, Règle d'inférence, Satisfiabilité.
- Model-checking: Système de transitions, Logique temporelle.
- Analyse statique: Correction de l'analyse, Fausse-alarme.
- Système de type: Typage statique, Sûreté du typage.
- Raisonnement déductif: Pre/Post-condition, Invariant de boucle.
- Assistant de preuves.