Contexte

- Financement FNRAE
- Partenaires: IRIT, ONERA
- Validation code C embarqué: très répandu, normes strictes (processus, codage, validation)
- Analyse statique: interprétation abstraite, méthode déductive
- Prototype yasa

Objectifs

- **ne pas** écrire de domaines (ou analyses) spécifiques
- combinaison d'analyses statiques et/ou tests dynamiques
- conception et programmation propres:

→ modularité, généricité, correction, pédagogie

- intégration dans l'environnement frama-C / CIL
- intégration dans les processus industriels de validation

Calendrier

- T0 = Janvier 2009
- T0+6 (phase 1): D1 État de l'art sur la combinaison d'analyses statiques
- T0+18 (phase 1): D2 Collaboration d'analyses pour la vérification d'erreurs à l'exécution

Calendrier

- T0 = Janvier 2009
- T0+6 (phase 1): D1 État de l'art sur la combinaison d'analyses statiques
- T0+18 (phase 1): D2 Collaboration d'analyses pour la vérification d'erreurs à l'exécution
- T0+24: Bilan de l'application de la méthode proposée en phase 1 au cas d'étude

Calendrier

- T0 = Janvier 2009
- T0+6 (phase 1): D1 État de l'art sur la combinaison d'analyses statiques
- T0+18 (phase 1): D2 Collaboration d'analyses pour la vérification d'erreurs à l'exécution
- T0+24: Bilan de l'application de la méthode proposée en phase 1 au cas d'étude
- T0+36 (phase 2): D1 Collaboration d'analyses pour la vérification de propriétés fonctionnelles
- T0+36 (phase 2): D2 Intégration de la collaboration d'analyses au processus industriel

Interprétation Abstraite

Sémantique concrète:

1. on définit le domaine sémantique concret D. Par exemple, dans la sémantique collectrice:
   $D \triangleq \mathit{Event}^\star \times \mathit{GlobalEnv} \times \mathit{CallStack}$
2. on exprime la sémantique (collectrice) concrète du programme par (le *lfp* d'une) fonctionnelle monotone F.
3. on définit la propriété $P \subseteq 2^D$ à valider.

Interprétation Abstraite

Sémantique concrète:

1. on définit le domaine sémantique concret D. Par exemple, dans la sémantique collectrice:
   $D \triangleq \mathit{Event}^\star \times \mathit{GlobalEnv} \times \mathit{CallStack}$
2. on exprime la sémantique (collectrice) concrète du programme par (le *lfp* d'une) fonctionnelle monotone F.
3. on définit la propriété $P \subseteq 2^D$ à valider.

- la sémantique collectrice est le plus souvent utilisée.
- d'autres sémantiques sont possibles (dénotationnelle ?).
- F peut représenter une propagation de propriétés en marche **avant** ou arrière.

Interprétation Abstraite

Sémantique abstraite:

1. on cherche un domaine abstrait (relationnel ou non) $D^\#$ qui capture le plus précisément P.
2. on définit la correspondance de Galois:
   $2^D \xrightarrow{\alpha} D^\#$ et/ou $2^D \xleftarrow{\gamma} D^\#$.
3. on définit $F^\#$ monotone tel que: $\alpha(F(d)) \sqsubseteq F^\#(\alpha(d))$.
4. on calcule un point fixe de $F^\#$ (le *lfp*, sauf élargissement).
5. on teste $\alpha(\mathit{lfp}\, F) \sqsubseteq \mathit{lfp}\, F^\# \overset{?}{\sqsubseteq} \alpha(P)$

Interprétation Abstraite

Sémantique abstraite:

1. on cherche un domaine abstrait (relationnel ou non) $D^\#$ qui capture le plus précisément P.
2. on définit la correspondance de Galois: $2^D \xrightarrow{\alpha} D^\#$ et/ou $2^D \xleftarrow{\gamma} D^\#$.
3. on définit $F^\#$ monotone tel que: $\alpha(F(d)) \sqsubseteq F^\#(\alpha(d))$.
4. on calcule un point fixe de $F^\#$ (le *lfp*, sauf élargissement).
5. on teste $\alpha(\mathit{lfp}\, F) \sqsubseteq \mathit{lfp}\, F^\# \stackrel{?}{\sqsubseteq} \alpha(P)$

- on obtient une sur-approximation (un post-point fixe) du *lfp* de F.
- une sous-approximation est possible, mais peu utilisée.
- on peut utiliser le *gfp* sur $F^\#$.

Méthode Déductive

1. méthode syntaxique:

$\longrightarrow$ langage de propriétés, syntaxe des programmes.

2. en marche arrière (pas calculable en marche avant).
3. une règle d'inférence pour:

   chaque connecteur logique, chaque construction de programme.
4. on applique ces règles, pour une postcondition et un programme donné.
5. on obtient une formule représentant une précondition.

Méthode Déductive

- exemple de l'affectation:

$$\frac{}{\{[exp \mid x]Post\}x := exp\{Post\}}$$

- exemple de la boucle (correction partielle):

$$\frac{\{Inv \wedge Cond\}P\{Inv\}}{\{Inv\}\texttt{while}(Cond)\,P;\{Inv \wedge \neg Cond\}}$$

Méthode Déductive

- exemple de l'affectation:

$$\frac{}{\{[exp \mid x]Post\}x{:=}exp\{Post\}}$$

- exemple de la boucle (correction partielle):

$$\frac{\{Inv \wedge Cond\}P\{Inv\}}{\{Inv\}\texttt{while}(Cond)\,P;\{Inv \wedge \neg Cond\}}$$

- méthode exacte (car symbolique), mais coûteuse.
- calcul de plus faible précondition (modulo les invariants de boucle).

Test Dynamique

1. créer un jeu de scénarios de test (exécutions) à partir des exigences fonctionnelles.

$\longrightarrow$ long, difficile (il faut trouver les bonnes valeurs).

2. le jeu de tests doit satisfaire un critère de couverture structurelle:

   blocs, enchaînements, MC/DC, ...

3. la couverture structurelle concerne le code source ou le binaire généré par le compilateur.
4. cette couverture constitue le test d'arrêt de la génération de nouvelles exécutions.

Propriétés

- Runtime Errors: débordement arithmétique, division par zéro, erreur de casting, pointeur invalide.
- fonctionnelles: assertion arbitraire sur les variables (à un point de programme donné).
- autres propriétés:

→ comportementale: enchaînement des calculs, des appels, ...

→ couverture: pour éviter le code mort.

→ temporisée: calcul de WCET, ...

Langage de propriétés

- langage généraliste (pour C): arithmétique, quantification, pointeur, bit-level ?
- langage pivot entre les différentes approches:

  traduction propriété $\leftrightarrow$ élément abstrait.
- pas de modèle mémoire spécifique (interprétation des pointeurs).

$\rightarrow$ ACSL (langage de spec. de caveat).

Lots fonctionnels

- morceau de code autonome: procédure, corps de boucle, bloc séquentiel.
- spécifié aux frontières ($\sim$ triplet de Hoare).
- on cherche parfois à abstraire certaines variables $\vec{x}$.

$\longrightarrow$ globales, paramètres, locales, indices de boucle, ...

$$\{Input\}$$
$$Lot[\vec{x}]$$
$$\{Output\}$$

Modularité

Permettre la composition de domaines:

- domaines (treillis) de base:

→ arithmétique entière: intervalles, modulo, ...

→ arithmétique flottante: ?

→ types finis, fonctions statiques: sous-ensembles, ...

→ pointeurs, tableaux: alias, modèle mémoire, ...

- constructeurs simples de domaines: produit (cartésien, réduit), somme.
- structures de données: pile, tableau, environnement, ...

Modularité

- un cadre théorique et pratique (en OCAML) pour la composition de domaines.
- contructions catégoriques ($\sim$ catégorie des domaines abstraits).

```
module type DOM =
sig
  include DATA
  type base
  val leq       : t -> t -> bool
  val bot       : t
  val top       : t
  val union     : t -> t -> t
  val widen     : t -> t -> t
  val inter     : t -> t -> t
end
```

Modularité

Structures de données exprimées par foncteurs et monades.

```
module type DOM_FUNCTOR =
  sig
    module Object: functor (D: DOM) -> DOM
    module Arrow : functor (D: DOM) -> functor (D': DOM) ->
      sig
        val map: (D.t->D'.t) -> Object(D).t -> Object(D').t
(...)
module type DOM_MONAD =
sig
  module F: DOM_FUNCTOR
  module M: functor (D: DOM) ->
    sig
      val unit: D.t -> F.Object(D).t
      val join: F.Object(F.Object(D)).t -> F.Object(D).t
(...)
```

Modularité

Opérations sur les données exprimées par transformations naturelles.

```
module type DOM_NATURAL =
sig
  module F: DOM_FUNCTOR
  module G: DOM_FUNCTOR

  module T: functor (D: DOM) ->
    sig
      val apply: F.Object(D).t -> G.Object(D).t
    end
end
```

Modularité

Exemple de la pile générique.

- une pile est une séquence:

$\longrightarrow$ foncteur: $D \overset{Pile}{\Rightarrow} D^{\star}$

$\longrightarrow$ foncteur: $D \overset{PileExpanse}{\Longrightarrow} 1 + D \times D^{\star}$

$\longrightarrow$ transformation naturelle: $Pile(D) \overset{pop}{\rightarrow} PileExpanse(D)$

$\longrightarrow$ transformation naturelle: $PileExpanse(D) \overset{push}{\rightarrow} Pile(D)$

Modularité

Exemple de la pile générique.

- une pile est une séquence:

→ foncteur: $D \overset{Pile}{\Rightarrow} D^\star$

→ foncteur: $D \overset{PileExpanse}{\Longrightarrow} 1 + D \times D^\star$

→ transformation naturelle: $Pile(D) \overset{pop}{\rightarrow} PileExpanse(D)$

→ transformation naturelle: $PileExpanse(D) \overset{push}{\rightarrow} Pile(D)$

- Pour un domaine abstrait $D^\# \approx 2^D$:

→ $2^{PileExpanse(D)} \approx 2 \times (D^\# \rightarrow 2^{Pile(D)})$

→ nouvelle transformation: $2^{Pile(D)} \overset{pop^\#}{\rightarrow} 2^{PileExpanse(D)}$

→ nouvelle transformation: $2^{PileExpanse(D)} \overset{push^\#}{\rightarrow} 2^{Pile(D)}$

Prototype YASA

- prototype basé sur l'approche de l'interprétation abstraite.
- utilisé en enseignement (M2).
- implanté en OCAML.
- l'infrastructure est définie selon nos principes catégoriques.
- utilisation massive des modules et foncteurs OCAML.

Prototype YASA

- prototype basé sur l'approche de l'interprétation abstraite.
- utilisé en enseignement (M2).
- implanté en OCAML.
- l'infrastructure est définie selon nos principes catégoriques.
- utilisation massive des modules et foncteurs OCAML.
- tous les domaines nécessaires ne sont pas encore implantés ou portés. Pour l'instant:

→ arithmétique, appels de fonctions.

→ opérateurs avant et arrière.

→ *lfp* et *gfp*.

Intégration dans Frama-C

Travaux en cours.

- notre prototype est un plugin Frama-C.
- domaines exportés dans un format standard XML.
- fonctionnalités load/store.

Intégration dans Frama-C

Travaux en cours.

- notre prototype est un plugin Frama-C.
- domaines exportés dans un format standard XML.
- fonctionnalités load/store.

- il existe un plugin Eclipse !

Intégration des autres approches

Travaux en cours.

- méthode déductive (dans le cadre théorique et dans l'outil).

$\longrightarrow$ combinaison marche avant et arrière sur un même lot.

Intégration des autres approches

Travaux en cours.

- méthode déductive (dans le cadre théorique et dans l'outil).

$\longrightarrow$ combinaison marche avant et arrière sur un même lot.

- tests dynamiques et couverture structurelle:
  1. chemins intéressants définis par expressions régulières (sur les gardes).
  2. *slicing* de l'interprétation abstraite du programme C selon ces chemins.
  3. treillis des chemins pour sauver ou recharger chemins déjà calculés.

Modularité

Travaux de Francesco Logozzo. Contexte général:

- Combinaison d'analyses (éventuellement $\neq$) inter-lots.
- réglage du ratio précision/complexité.
- interprétation abstraite pure (sur-approximation de la sémantique).

Modularité

Travaux de Francesco Logozzo. Contexte général:

- Combinaison d'analyses (éventuellement $\neq$) inter-lots.
- réglage du ratio précision/complexité.
- interprétation abstraite pure (sur-approximation de la sémantique).
- instancié au langage à objets JAVA, pour de l'analyse avant.

Modularité

Travaux de Francesco Logozzo. Méthode de calcul:

1. on fournit une spécification aux frontières inter-lots qui soit bien invariante.
   - pas par point fixe ! Dans le cas de JAVA, par typage.
   - si on part d'une pré-condition d'un lot et qu'on l'exécute, on vérifie sa post-condition.
   - les erreurs (absence non garantie par typage) ne se propagent pas.

Modularité

Travaux de Francesco Logozzo. Méthode de calcul:

1. on fournit une spécification aux frontières inter-lots qui soit bien invariante.
   - pas par point fixe ! Dans le cas de JAVA, par typage.
   - si on part d'une pré-condition d'un lot et qu'on l'exécute, on vérifie sa post-condition.
   - les erreurs (absence non garantie par typage) ne se propagent pas.
2. on calcule le *gfp* de la fonctionnelle qui, à chaque pré-condition d'un lot, calcule sa post-condition.
3. pour chaque lot, cette fonctionnelle:
   - traite une sémantique abstraite particulière.
   - calcule le *lfp* des instructions du lot, partant de la pré-condition.
   - les spécifications s'affinent au fur et à mesure.

Combinaison IA / méthode déductive

Travaux de Yannick Moy. Contexte général.

- on considère une spécification, soit dans une boucle, soit derrière une boucle.
- on veut calculer une pré-condition (pas trop forte), avant la boucle, indépendante des itérations de cette boucle.
- on utilise une première passe d'IA pour déterminer des invariants sur les indices de boucle.
- généralisable: appels de fonctions, plusieurs spec.

Exemple:

$$\{Input\}$$
$$\texttt{while}(Cond)$$
$$Boucle$$
$$\{Output\}$$

Combinaison IA / méthode déductive

Travaux de Yannick Moy. Méthode de calcul.

- on détermine les variables changeantes de la boucle.
- on calcule la première passe d'IA.

$$\{\dots\}$$
$$\texttt{while}(Cond)$$
$$\{Inv_L\}$$
$$Boucle[\vec{x}]$$
$$\{Inv_O, Output\}$$

Combinaison IA / méthode déductive

Travaux de Yannick Moy. Méthode de calcul.

- on détermine les variables changeantes de la boucle.
- on calcule la première passe d'IA.

$$\{\ldots\}$$
$$\texttt{while}(Cond)$$
$$\{Inv_L\}$$
$$Boucle[\vec{x}]$$
$$\{Inv_O, Output\}$$

- on a: $Input \triangleq \forall \vec{x}.(Inv_L \wedge \pm Cond) \Rightarrow \mathcal{WP}(Inv_O \Rightarrow Output)$

Combinaison IA / tests dynamiques

Travaux de Xavier Rival. Contexte général:

- on suppose une spécification violée en un point de programme.
- but: reconstruire un véritable scénario de test menant à l'erreur ou prouver l'absence de scénario.
- malgré les sur-approximations dûes à l'IA.
- marche arrière: de l'erreur jusqu'au point de programme initial.
- on considère les points de programme intermédiaires testant les conditions (conditionnelles et boucles).

Combinaison IA / tests dynamiques

Travaux de Xavier Rival. Contexte général:

- on suppose une spécification violée en un point de programme.
- but: reconstruire un véritable scénario de test menant à l'erreur ou prouver l'absence de scénario.
- malgré les sur-approximations dûes à l'IA.
- marche arrière: de l'erreur jusqu'au point de programme initial.
- on considère les points de programme intermédiaires testant les conditions (conditionnelles et boucles).
- très expérimental !

Partionnement de traces

Travaux de Xavier Rival et Laurent Mauborgne. Contexte général:

- les traces du flôt de contrôle fournissent un moyen simple de partitionner.
- on peut inclure la pile d'appels (pas de récursivité).
- on abstrait ces traces par des ensembles.
- on abstrait ces ensembles.

Partionnement de traces

Travaux de Xavier Rival et Laurent Mauborgne. Contexte général:

- les traces du flôt de contrôle fournissent un moyen simple de partitionner.
- on peut inclure la pile d'appels (pas de récursivité).
- on abstrait ces traces par des ensembles.
- on abstrait ces ensembles.

Exemple:

```
1: int x, sgn;
2: if (x < 0)
3:   sgn = -1;
   else
4:   sgn = 1;
5: y = x / sgn;
```

$\{1,2,3,5\} \mapsto x < 0, sgn = -1, y = -x;$
$\{1,2,4,5\} \mapsto x \geq 0, sgn = 1, y = x$

Autres perspectives

- pile d'appels.
- méthode CEGAR (abstraction de prédicats, travaux de Thomas Ball).
- gestion mémoire fine (stockage des calculs).