Vérification de propriétés LTL sur des programmes C par génération d'annotations

Travail partiellement supporté par le projet PFC de System@tic.

Julien Groslambert[a] Nicolas Stouls[b]

[a] Trusted-Labs
Julien.Groslambert@trusted-labs.com

[b] Projet ProVal, INRIA Saclay – Île-de-France
[b] INSA Lyon, CITI
Nicolas.Stouls@inria.fr

25 novembre 2008

Introduction

- Contexte :
  - Propriétés temporelles
  - Génération d'annotations de programmes
- Extension de travaux existants :
  - **Huisman et Trentelman**
    Génération d'annotations JML à partir de propriétés JTPL
    Exemple de limitation JTPL : $(x = 0) \implies \circ(y = 3)$
  - **Groslambert**
    Extension à la LTL + Preuve de correction
    Orienté validation par animation
- Limitations pour la preuve :
  - Insuffisances des hypothèses
  - Explosion combinatoire des OP (nombre et complexité)
- Notre objectif : Favoriser l'automatisation de la preuve
  Pour l'instant : Focalisation sur l'aspect sûreté

Plan

1. Exemple fil rouge
2. Rappel [J. Groslambert] : conditions suffisantes
3. Démarche :
   1. *Ajout d'hypothèses*
   2. *Propagation statique des contraintes*
4. Approche outillée
5. Conclusion

Exemple fil rouge

Transaction en deux parties

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @ 0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @ ∧ (cpt = 0 ⇒ status = 0); */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Granularité de l'observation/vérification : appel/retour d'opérations

Exemple fil rouge

Transaction en deux parties

```
int cpt=3 ;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3 ; */

int status=0 ;
/*@ global invariant invst :
    0 ≤ status ≤ 1 ; */

//@ ensures 0 ≤ \result ≤ 1 ;
int init() {...}

//@ ensures 0 ≤ \result ≤ 1 ;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1 ;
int main(){
  /*@ loop invariant i :
    @ 0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @ ∧ (cpt = 0 ⇒ status = 0); */
  while (cpt>0) {
    status=init() ;
    if (status && commit()) goto label_ok ;
    cpt-- ;
  }
  return 0 ;
  label_ok : return 1 ;
}
```

Granularité de l'observation/vérification : appel/retour d'opérations

Exemple fil rouge

Transaction en deux parties

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @ 0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @ ∧ (cpt = 0 ⇒ status = 0); */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Granularité de l'observation/vérification : appel/retour d'opérations

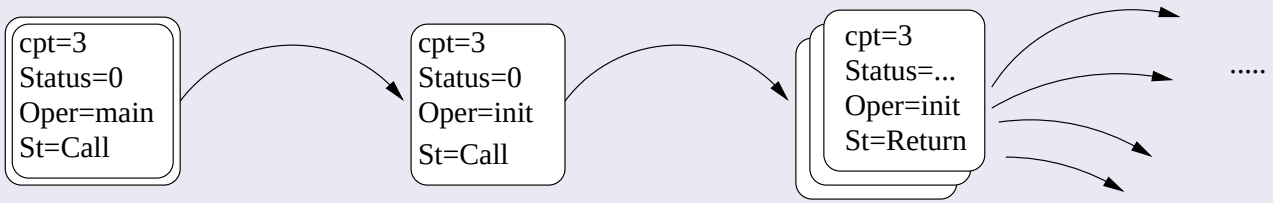
Exemple fil rouge

Transaction en deux parties

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @ 0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @ ∧ (cpt = 0 ⇒ status = 0); */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Granularité de l'observation/vérification : appel/retour d'opérations



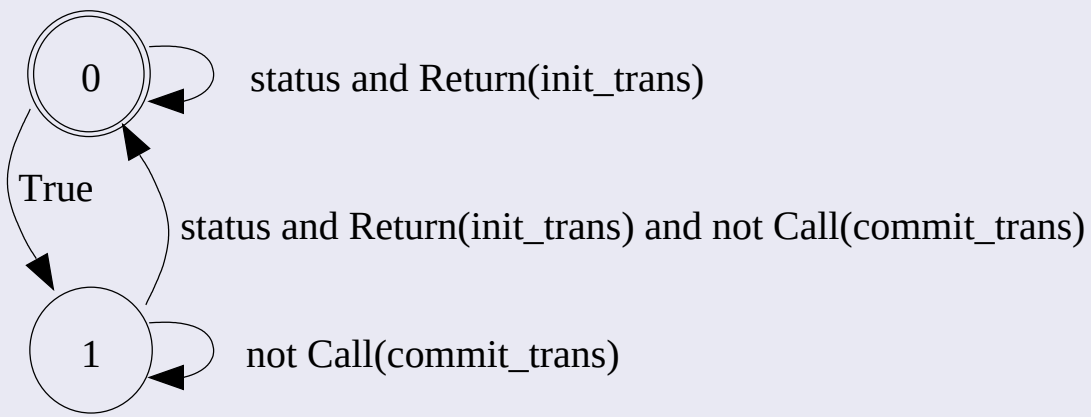
Exemple fil rouge

Propriété d'atomicité à vérifier

$$\Box((\neg \textbf{RETURN}(\textit{init}) \ \vee \ \neg \textit{status}) \Rightarrow \bigcirc \neg \textbf{CALL}(\textit{commit}))$$

- Appel de *commit* seulement si *init* est appelée juste avant et sans erreur
- Propriété porte sur les *opérations* **et** les *variables* du programme

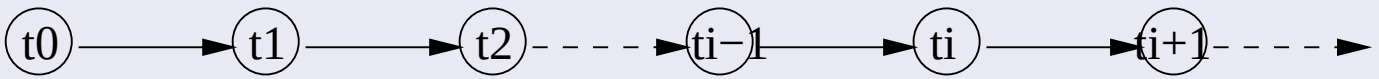
Automate de Büchi de sûreté de la propriété *(Retour 12)*



Correction programme VS formule LTL

Principe

- Programme : ensemble $\text{PATH}_{\text{Prog}}$ de traces d'exécution



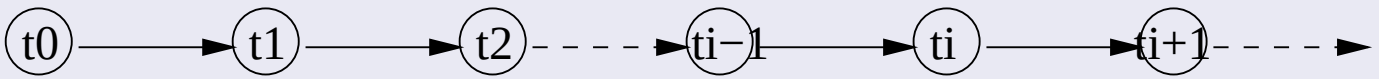
- LTL : ensemble $\text{PATH}_{\text{Büchi}}$ de chemins de l'automate de Büchi
- Vérification basée sur la synchronisation programme/LTL :

$$\forall t \in \text{PATH}_{\text{Prog}} \cdot \exists c \in \text{PATH}_{\text{Büchi}} \cdot \forall i \cdot t_i \models P_i(c)$$

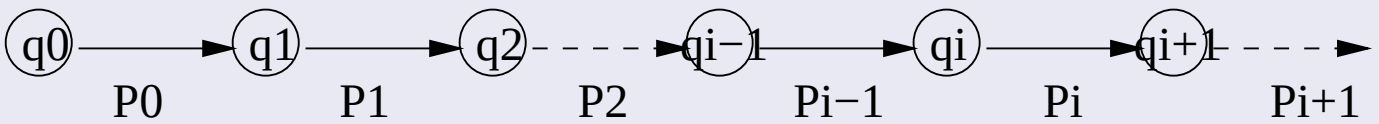
Correction programme VS formule LTL

Principe

- Programme : ensemble $\text{PATH}_{\text{Prog}}$ de traces d'exécution



- LTL : ensemble $\text{PATH}_{\text{Büchi}}$ de chemins de l'automate de Büchi



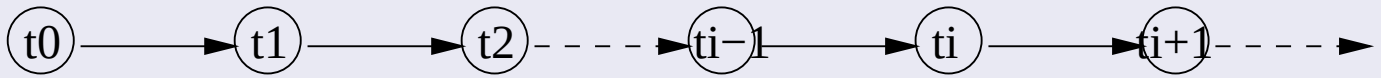
- Vérification basée sur la synchronisation programme/LTL :

$$\forall t \in \text{PATH}_{\text{Prog}} \cdot \exists c \in \text{PATH}_{\text{Büchi}} \cdot \forall i \cdot t_i \models P_i(c)$$

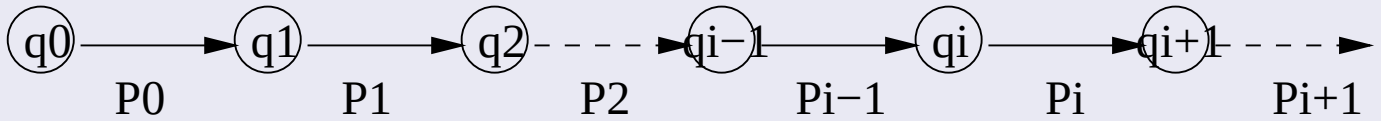
Correction programme VS formule LTL

Principe

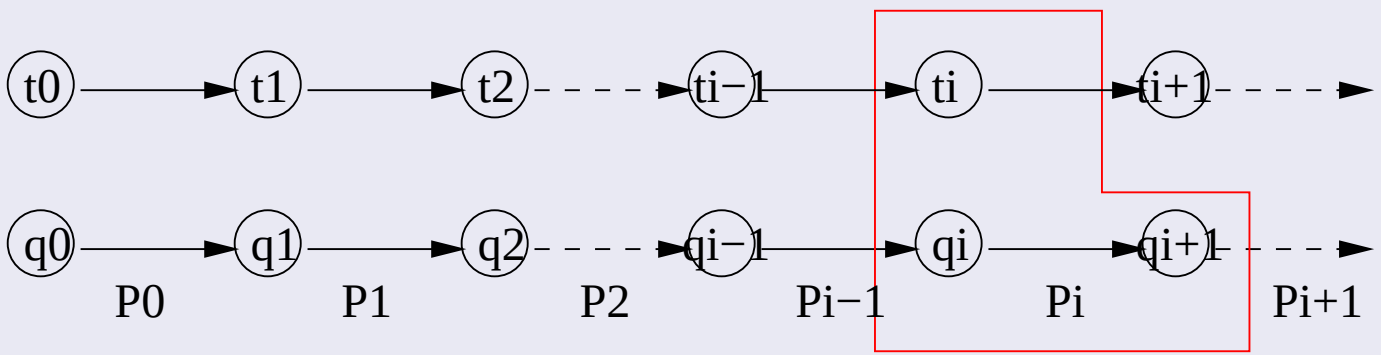
- Programme : ensemble $\text{PATH}_{\text{Prog}}$ de traces d'exécution



- LTL : ensemble $\text{PATH}_{\text{Büchi}}$ de chemins de l'automate de Büchi



- Vérification basée sur la synchronisation programme/LTL :



$$\forall t \in \text{PATH}_{\text{Prog}} \cdot \exists c \in \text{PATH}_{\text{Büchi}} \cdot \forall i \cdot t_i \models P_i(c)$$

Synchronisation Büchi-Prog

Definition (Fonction de synchronisation)

$A = \langle Q, q_0, R \rangle \in \text{BUCHI}$ $\quad \sigma \in \text{PATH}_{\text{Prog}}$

$\text{sync} : \text{BUCHI} \times \text{PATH} \times \mathbb{N} \to 2^Q$ est définie par :

- $\text{sync}(A, \sigma, 0) = \{q_0\}$
- Pour tout $i > 0$:

$$\text{sync}(A, \sigma, i) = \left\{ q' \;\middle|\; \begin{array}{c} \exists \langle q, P, q' \rangle \in R \cdot \wedge \\ \sigma_{(i-1)} \models P \wedge \\ q \in \text{sync}(A, \sigma, i-1) \end{array} \right\}$$

Definition (Condition d'acceptation)

(C_{Sync}) $\quad \forall i \in 0..(\text{len}(\sigma) - 1) \cdot \text{sync}(A, \sigma, i) \neq \emptyset$

Synchronisation Büchi-Prog

Definition (Fonction de synchronisation)

$A = \langle Q, q_0, R \rangle \in \text{BUCHI}$ $\quad \sigma \in \text{PATH}_{\text{Prog}}$

$\text{sync} : \text{BUCHI} \times \text{PATH} \times \mathbb{N} \to 2^Q$ est définie par :

- $\text{sync}(A, \sigma, 0) = \{q_0\}$
- Pour tout $i > 0$:

$$\text{sync}(A, \sigma, i) = \left\{ q' \;\middle|\; \begin{array}{c} \exists \langle q, P, q' \rangle \in R \cdot \wedge \\ \sigma_{(i-1)} \models P \wedge \\ q \in \text{sync}(A, \sigma, i-1) \end{array} \right\}$$

Definition (Condition d'acceptation)

(C_{Sync}) $\qquad \forall i \in 0..(\text{len}(\sigma) - 1) \cdot \text{sync}(A, \sigma, i) \neq \emptyset$

Annotations de synchronisation

Annotations de synchronisation [Thèse de J. Groslambert]

Annotations à générer :

- CLAUSE_{Decl_A} Déclarations initiales
- CLAUSE_{Trans_A} Affectations à chaque appel/retour d'opération
- CLAUSE_{Sync_A} Invariants

On ne considère pas ici $\text{CLAUSE}_{Büchi_A}$ (Vivacité)

avec

$$\text{CLAUSE}_{Decl_A} = \bigcup_{q \in Q} \left\{ \begin{array}{lll} \mathcal{D}(v_q, \text{true}) & si & q = q_0 \\ \mathcal{D}(v_q, \text{false}) & si & q \neq q_0 \end{array} \right\}$$

$$\text{CLAUSE}_{Trans_A} = \bigcup_{q \in Q} \{\mathcal{A}(v_q, \exists (q', P, q) \in R \cdot (P \wedge v_{q'}))\}$$

$$\text{CLAUSE}_{Sync_A} = \{\mathcal{I}(\bigvee_{q \in Q}(v_q))\}$$

$\mathcal{I}(\mathtt{I})$:	invariant $(\forall i.\sigma(i) \models \mathtt{I})$
$\mathcal{D}(v, Init)$:	déclaration du ghost v avec $Init$ comme valeur initiale.
$\mathcal{A}(v, E)$:	assignation du ghost v par E *(E : sans effet de bord)*.

Application à la vérification par la preuve

Forme générale des OP générées

$$
\begin{array}{ll}
H_1 & \bigvee_{i=0}^{NbStates} \backslash\mathbf{old}(curStates[i]) \neq 0 \\
H_{2\ldots n} & \text{/* Hypothèses issues de } \mathrm{CLAUSE}_{Trans_A} \text{ */} \\
H_{n+1\ldots m} & \text{/* Hypothèses liées au corps de l'opération */} \\
\hline
But & \bigvee_{i=0}^{NbStates} curStates[i] \neq 0 \text{ /* } \mathrm{CLAUSE}_{Sync_A} \text{ */}
\end{array}
$$

Difficultés liées à la preuve automatique

- H_1 trop faible
  - Aucune transition depuis $q_i \Rightarrow$ montrer que q_i non synchronisé précédemment
  - $H_{n+1\ldots m}$ difficilement exploitables
- Pas de lien entre états programme/états l'automate

Application à la vérification par la preuve

Forme générale des OP générées

$$
\begin{array}{ll}
H_1 & \bigvee_{i=0}^{NbStates} \backslash\mathbf{old}(curStates[i]) \neq 0 \\
H_{2\ldots n} & \text{/* Hypothèses issues de } \mathrm{CLAUSE}_{Trans_A} \text{ */} \\
H_{n+1\ldots m} & \text{/* Hypothèses liées au corps de l'opération */} \\
\hline
But & \bigvee_{i=0}^{NbStates} curStates[i] \neq 0 \text{ /* } \mathrm{CLAUSE}_{Sync_A} \text{ */}
\end{array}
$$

Propositions

- Créer des liens Büchi ($H_{n+1\ldots m}$) – Prog ($H_{2\ldots n}$)
  ($H_{m+1\ldots k}$: Spécification de l'automate et de ses transitions)
- Limiter les disjonctions *But* et H_1
  (Analyse statique des états inatteignables)

Création de liens Büchi-Prog

Principe

- Axiomatisation de l'automate
- Mémorisation des transitions franchies
- Ajout d'invariants de liaison automate-programme

Mise en œuvre

Création de liens Büchi-Prog

Principe

- Axiomatisation de l'automate
- Mémorisation des transitions franchies
- Ajout d'invariants de liaison automate-programme

Mise en œuvre

```
/*@ logic integer transStart(integer tr) ;
  @ axiom transStart_0 : transStart(0) = /* État de départ de la transition 0 */ ;
  @ axiom transStart_1 : transStart(1) = /* État de départ de la transition 1 */ ;

  @ logic integer transStop(integer tr) ;
  @ axiom transStop_0 : transStop(0) = /* État d'arrivée de la transition 0 */ ;
  @ axiom transStop_1 : transStop(1) = /* État d'arrivée de la transition 1 */ ;

  @ predicate transCond(integer tr)=
  @   (tr = 0 ⇒ .../* Condition de la transition 0 */) ∧
  @   (tr = 1 ⇒ .../* Condition de la transition 1 */) ∧    */
```

Création de liens Büchi-Prog

Principe

- Axiomatisation de l'automate
- Mémorisation des transitions franchies
- Ajout d'invariants de liaison automate-programme

Mise en œuvre

- Ajout d'une variable *curTrans*
- Redondance partielle avec *curStates* *(simplification des OP)*
- Cohérence *curTrans* / *curStates* préservée par invariant

Création de liens Büchi-Prog

Principe

- Axiomatisation de l'automate
- Mémorisation des transitions franchies
- Ajout d'invariants de liaison automate-programme

Mise en œuvre

/*@ **global invariant** Non-atteignabilité$_1$:
@ $\forall st; 0 \leq st < NbStates \wedge$
@ $\left(\begin{array}{l} \forall tr; 0 \leq tr < NbTrans \\ \quad \Rightarrow curTrans[tr] = 0 \ \vee \ transStop(tr) \neq st \ \vee \\ \quad\quad \neg transCond(tr) \ \vee \ curStates_{Old}[transStart(tr)] = 0 \end{array} \right)$
@ $\Rightarrow curStates[st] = 0;$
*/

Ajout de spécification aux opérations

- Restriction des états/transitions pour chaque opération
  - *Dans un premier temps : manuelle*
  - *Dans la suite de l'exposé : génération automatique*
  - *Remplace et affine l'invariant de synchronisation*

- Avantage :

  Restriction des disjonctions H_1 et But

- Inconvénient :

  Nouvelles OP à chaque appel/retour

Première approximation des pré/post-conditions

- Sur-approximation depuis la propriété :

Objectif : Retrait des transitions/états directement interdits

- Gardes des transitions :
  - Abstraction des expressions (sur variables)
  - Exploitation des prédicats (sur nom d'opérations)

Ex. *Call*(*commit*) : ne peut franchir que la transition allant de 0 vers 1 (Fig. 5)

- Pré-condition de *main* : état initial de la propriété

Algo de construction pour chaque opération *Op*

$Pré(Op).trans \leftarrow \{tr \mid \text{l'appel d'}Op \text{ n'est pas interdit par } tr\}$
$Pré(Op).état \leftarrow transStop[Pré(Op).trans]$

$Post(Op).trans \leftarrow \{tr \mid \text{le retour d'}Op \text{ n'est pas interdit par } tr\}$
$Post(Op).état \leftarrow transStop[Post(Op).trans]$

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

- Propagation des spécifications approximées
- Abstraction des expressions
- IA avant : limitation des post-conditions aux états atteignables
  $Post(Op) \leftarrow Post(Op) \cap Fwd(Pré(Op), body(Op))$
- IA arrière : limitation des pré-conditions aux états non-divergents
  $Pré(Op) \leftarrow Pré(Op) \cap Backward(Post(Op), body(Op))$

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

$Fwd(P, [\,]) \triangleq P$

$Fwd(P, (x = E) :: L) \triangleq Fwd(P, L)$

$Fwd(P, Call(Op) :: L) \triangleq Fwd(Post(Op), L)$

$Fwd(P, (\texttt{IF}\ (c)\ B_1\ \texttt{ELSE}\ B_2) :: L) \triangleq$
$\quad$ **let** $p_1, p_2 = Fwd(P, B_1), Fwd(P, B_2)$ **in** $Fwd(p_1 \ \vee\ p_2, L)$

$Fwd(P, (\texttt{return}\ e) :: []) \triangleq$
$\quad (trans = \{tr \mid transStart(tr) \in P.état \wedge$ le retour d'Op peut franchir $tr\}$,
$\quad état = transStop[Post(Op).trans])$

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

- A chaque passe : tous les appels sont observés
- Recensement des états d'appel de chaque opération
- Restriction des specs aux cas d'utilisation
- Point fixe des spécifications termine

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

- Problèmes :
  - Propager les pré/post à travers la boucle
  - Annoter la boucle pour rendre la vérification possible

    (Induction d'un invariant en terme des états/transitions de l'automate)

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

Forme générale d'une boucle dans Frama-C :

$\{pré_{boucle}\}$
while(1) {
  $\{pré_{corps}\}$
  if(c) **goto** LabelEndLoop ;
  . . . /* *Corps de la boucle* */
  $\{post_{corps}\}$
}
LabelEndLoop :
$\{post_{boucle}\}$

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

- Pré/post de la boucle connus
- Construction des pré/post du corps par IA avant-arrière
- Propagation des contraintes du corps vers la spec de boucle
- Invariant construit :

  II@ **Loop invariant** i : $(Init \Rightarrow Pré_{boucle}) \wedge (\neg Init \Rightarrow Post_{corps})$

  Init : variable fraîche identifiant la première itération.

Propagation statique des contraintes

Différentes méthodes mises en œuvre

1. Interprétation abstraite avant-arrière
2. Restriction aux cas d'utilisation
3. Spécification des boucles
4. Raffinement des post-conditions

Mise en œuvre

- Post-conditions en terme des états d'entrée
- Exemple :

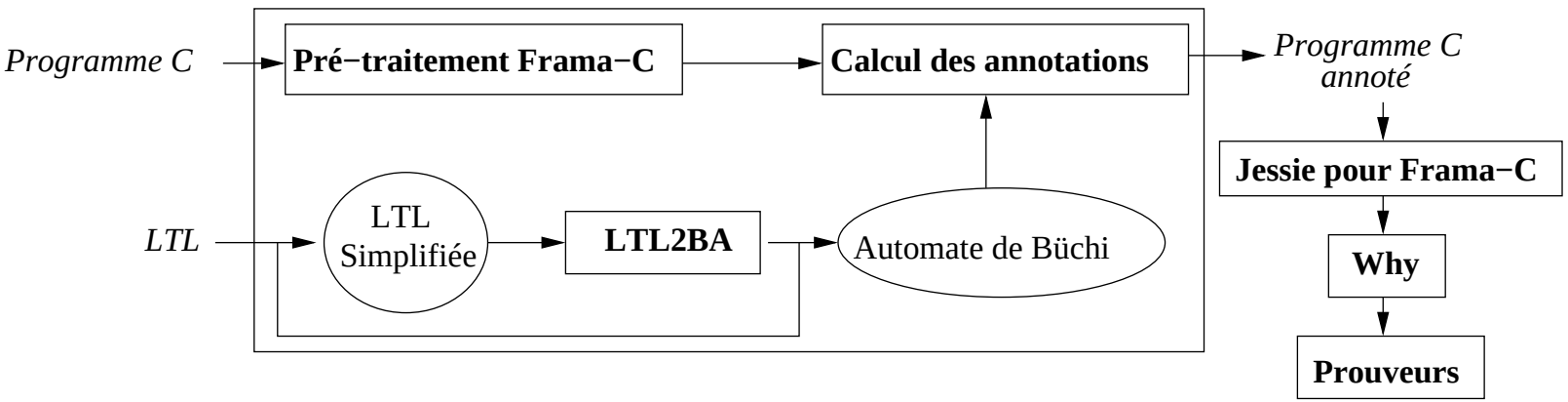
behavior buch_0 :
  assumes *curStates*[0] $\neq 0$
  ensures . . . */* Post-condition liée à l'état d'entrée 0 */*
behavior buch_1 :
  assumes *curStates*[1] $\neq 0$
  ensures . . . */* Post-condition liée à l'état d'entrée 1 */*

- Intérêt :

  Appels de fonctions depuis différents contextes

Plug-in Aoraï

- Implante les algos présentés
- Intégré dans Frama-C *(CEA-List + INRIA-Saclay)*
- Conversion LTL vers Büchi : LTL2BA *(LSV)*
- Génération OP+Preuve : Plug-in Jessie *(INRIA-Saclay)*
  - + Why *(INRIA-Saclay)*
  - + Simplify *(Compaq)*
  - + Alt-Ergo *(INRIA-Saclay)*
  - + . . .



Application au programme fil rouge

- Nombre d'OP générées sur l'exemple :
  Sans annotations générées : *62 OP*
  Avec annotations générées : *10174 OP (WP classique)*
  296 OP (FastWP)

- Fast WP : WP optimisé de Rustan Leino.
  (Un branchement dans le contrôle ne génère qu'une OP)
  Intéressant car beaucoup de variables instanciées

- 100% des OP vérifiées automatiquement.

Démo

Bilan

Résultats obtenus

- Extension de travaux en vérification de propriétés LTL
- Plus d'automatisation des preuves des OP liées à la sûreté
  - Ajout d'hypothèses de lien Automate-Programme
  - Simplifications statiques
- Validation de l'approche par le développement d'un outil

Comparaison avec l'approche de l'outil Jack

- Propagation des contraintes avant génération des annotations
  (Préservation de la sémantique des annotations)
- Propagation par IA (WP/SP abstrait)
  (L'approche Jack est une union des pré/post accessibles)

Jack : Générale *(Propage également les annotations manuelles)*
Aoraï : Ad-hoc *(Ajoute des annotations autour des existantes)*

Bilan

Résultats obtenus

- Extension de travaux en vérification de propriétés LTL
- Plus d'automatisation des preuves des OP liées à la sûreté
  - Ajout d'hypothèses de lien Automate-Programme
  - Simplifications statiques
- Validation de l'approche par le développement d'un outil

Comparaison avec l'approche de l'outil Jack

- Propagation des contraintes avant génération des annotations
  (Préservation de la sémantique des annotations)
- Propagation par IA (WP/SP abstrait)
  (L'approche Jack est une union des pré/post accessibles)

Jack : Générale *(Propage également les annotations manuelles)*
Aoraï : Ad-hoc *(Ajoute des annotations autour des existantes)*

Travaux futurs

- Renforcement de la précision des annotations
  - Ajout d'heuristiques d'IA (élargissement + graphe d'appel)
  - Diminuer l'abstraction de l'IA
  - Post-conditions sous hypothèses de valeurs des variables
- Étendre à la composante vivacité
  - Travaux initiaux prennent en compte la vivacité

Non dead-lock Pour un programme avec main : Post(main)$\subseteq$Acc
Non live-lock Exécution finie ou variant global

  - Exploiter les mécanismes de variant/decreases d'ACSL
  - Vérification sous hypothèses d'équité ?