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

Nicolas Stouls, Julien Groslambert

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

Nicolas Stouls *
(Nicolas.Stouls@insa-lyon.fr)
Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, France

Julien Groslambert
(Julien.Groslambert@trusted-labs.com)
Trusted-Labs
5 rue du Baillage, F-78000 Versailles

28 septembre 2008

Résumé

Ce travail propose une méthode de vérification de propriétés temporelles basée sur la génération automatique d'annotations de programmes. Les annotations générées sont ensuite vérifiées par des prouveurs automatiques *via* un calcul de plus faible précondition. Notre contribution vise à optimiser une technique existante de génération d'annotations, afin d'améliorer l'automatisation de la vérification des obligations de preuve produites.

Cette approche a été outillée sous la forme d'un plugin au sein de l'outil Frama-C pour la vérification de programmes C.

Mots clef : Preuve automatique, programmes C, propriétés LTL

1 Introduction et état de l'art

Propriétés temporelles Une branche importante de la vérification logicielle s'intéresse à la *vérification* de propriétés de sécurité de systèmes. Elle consiste à exprimer, dans une logique adéquate, les propriétés requises dans le cahier des charges, puis à la validation de celles-ci. La nature des propriétés que l'on souhaite vérifier peut-être très variée. Elle peut aller de la confidentialité des données (« Le code secret de la carte doit resté confidentiel »), au contrôle d'accès (« Seul un terminal identifié peut débiter la carte »). Nous nous intéressons ici à une classe de propriétés décrivant l'évolution du système au cours du temps : *les propriétés temporelles*.

Annotations de programme Pour vérifier le respect d'une spécification par un programme, il est possible de travailler au niveau du code source en insérant des annotations logiques (invariants et pré/post-conditions). Les

*Ce travail a été partiellement supporté par le projet PFC (Plate-Forme de Confiance) du pôle de compétitivité parisien System@tic.

travaux réalisés autour du langage d'annotations JML ont montré que celui-ci pouvait aisément combiner preuve formelle et validation à l'exécution [8]. Cependant, les langages d'annotations ne permettent généralement pas d'exprimer directement des propriétés temporelles. Représenter manuellement sous forme d'un ensemble d'annotations, une propriété temporelle, s'avère un travail fastidieux. De ce fait, il a été proposé de générer de façon systématique les annotations associées à une propriété temporelle.

État de l'art Une première proposition de génération automatique d'annotations à partir de propriétés temporelles a été proposée par Huisman et Trentelman [12]. Ces travaux se sont intéressés à la génération d'annotations JML à partir d'un langage de propriétés temporelles appelé JTPL. Ces travaux ont par la suite été étendus à la *logique temporelle linéaire*. Les fondations théoriques et la correction de la méthode ont été établis dans [7]. Ces travaux ont également donné lieu à l'implantation d'un prototype *JAG*, qui a permis d'expérimenter l'approche dans le cadre de la validation par test et par vérification par preuve.

L'approche de génération d'annotations à partir de propriétés temporelles s'est montrée pertinente dans le cadre de la vérification dynamique des annotations générées, notamment dans le cadre de la validation par le test. Par contre, la vérification statique par génération d'obligations de preuve s'est heurtée, dans la pratique, à deux problèmes majeurs : (i) bien que les annotations calculées soient correctes, elles ne sont parfois pas suffisantes pour résoudre les obligations de preuve générées ; (ii) les annotations générées entraînent une explosion combinatoire du nombre et de la complexité des obligations de preuves. Pour pallier a ces problèmes, l'utilisateur doit : (i) ajouter manuellement des invariants supplémentaires, faisant le lien entre le programme et la propriété. (ii) intervenir pour guider la preuve de façon interactive en sélectionnant les hypothèses nécessaires à la preuve. Globalement, ces étapes demandent une forte connaissance du système, de la propriété, du fonctionnement du prouveur et du mécanisme de génération d'annotations ;

Contributions À travers les travaux exposés dans ce papier, notre objectif est de favoriser l'automatisation du processus de vérification, en minimisant ces deux limitations. Pour cela, nous proposons de nouveaux algorithmes de génération d'annotations, permettant de générer une partie des hypothèses supplémentaires, qui étaient auparavant écrites manuellement, tout en optimisant le nombre et la forme des annotations générées.

Sommaire Dans un premier temps nous illustrons nos motivations en présentant un exemple fil rouge que nous déroulerons jusqu'au bout de cet article. Nous formalisons ensuite les langages considérés, avant de rappeler

brièvement quelques travaux existants et sur lesquels nous nous basons. Enfin, nous présentons nos contributions en sections 5, 6 et 7, avant de conclure.

2 Exemple fil rouge

```
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;
}
```

FIGURE 1 – Programme C décrivant une transaction en deux étapes

Pour illustrer l'approche décrite dans cet article nous utilisons le fragment de programme C présenté en figure 2. Ce dernier s'inspire du domaine des carte à puce et illustre l'une des problématiques de ces programmes critiques : réaliser une transaction par l'appel successif de deux opérations (*init* et *commit*). Le nombre d'essais est limité à 3 et la fonction renvoie une valeur différente de 0 si et seulement si les deux opérations on pu être appelées successivement sans erreur.

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

FIGURE 2 – Propriété d'atomicité à vérifier

Nous proposons de vérifier que ce programme respecte la formule décrite en figure 2. Celle-ci exprime une propriété d'atomicité : *commit* n'est appelé que si *init* a été appelé juste avant et a terminé sans erreur. Notons que la logique LTL utilisée a été étendue avec des prédicats (*Call* et *Return*) permettant d'exprimer le fait qu'une opération soit appelée ou s'arrête.

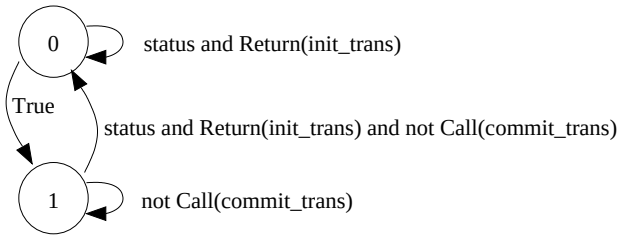

FIGURE 3 – Automate de Büchi de la propriété d'atomicité à vérifier

Pour effectuer la vérification, la méthode que nous utilisons [7], consiste à traduire la formule en un automate de Büchi (figure 3) qui sera ensuite réécrit sous la forme d'annotations dans le programme C. La traduction LTL vers Büchi est réalisée par l'outil LTL2BA [5] et ne sera pas détaillée ici.

3 Langage considéré

3.1 Sémantique de traces associée à un programme

La sémantique observationnelle d'un programme P peut être représentée par un système de transitions, dont les états sont des états « observables »du programme. Les états observables sont les états dans lesquel se trouve le système au moment d'un appel ou d'un retour d'opération. En plus des variables du systèmes ces états contiennent donc un couple de données supplémentaires : si c'est un appel ou un retour d'opération et le nom de l'opération concernée. Nous représentons ces informations avec deux variables `Oper` et `St` contenant respectivement le nom de l'opération et son statut (*Call* ou *Return*). Un extrait du système de transitions associé au programme fil rouge est décrit en figure 4.

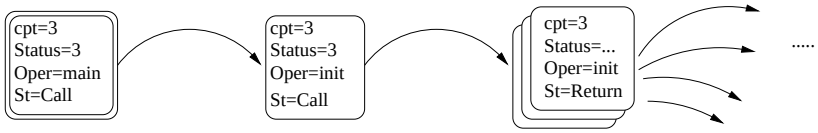

FIGURE 4 – Extrait du système de transitions associé au programme fil rouge

La sémantique d'un programme P peut être définie par l'ensemble des traces de son système de transitions. Intuitivement, une trace est la séquence des états d'un chemin du système de transitions, partant de l'état initial.

3.2 Sémantique des annotations de programme

Un programme peut être annoté par des assertions logiques. Celles-ci ne modifient pas son comportement (donc sa sémantique), mais permettent d'exprimer des propriétés que l'on veut vérifier à certains points du programme. Ici nous ne parlons que d'invariants, de pré-conditions et de post-conditions.

Définition 1 (Sémantique des assertions) *Nous définissons la sémantique des assertions d'un programme* P *par rapport à une trace* σ *de* P.

Annotation	*Sémantique*
$\mathcal{I}(\mathtt{I})$	$\forall i.\sigma(i) \models \mathtt{I}$
$\mathcal{P}\mathsf{recondition}(op, \mathtt{P})$	$\mathcal{I}(\mathtt{Oper} = op \ \wedge \ \mathtt{St} = \mathit{Call} \Rightarrow \mathtt{P})$
$\mathcal{P}\mathsf{ostcondition}(op, \mathtt{Q})$	$\mathcal{I}(\mathtt{Oper} = op \ \wedge \ \mathtt{St} = \mathit{Return} \Rightarrow \mathtt{Q})$

Nous ajoutons à cela les deux annotations suivantes, relatives aux variables d'annotations [1], qui sont des variables ne pouvant pas être utilisée par le programme, mais seulement dans les annotations :

- $\mathcal{D}(\mathtt{v}, \mathit{Init})$: déclaration de v avec *Init* comme valeur initiale.
- $\mathcal{A}(\mathtt{v}, \mathtt{E})$: assignation de v par E. L'expression E ne peut avoir d'effet de bord sur l'état du programme.

3.3 Sémantique des automates de Büchi de sûreté

Dans cette article, nous nous focalisons sur la composante sûreté des propriétés temporelles. C'est pourquoi, nous rappelons la définition de la structure d'automate de Büchi de sûreté et quelques résultats importants à propos de cette classe d'automates.

Définition 2 (Automate de Büchi de sûreté) *Soit* PRED *l'ensemble des prédicats sur les états du programme. Un automate de Büchi défini sur* PRED *est un triplet* $\langle \mathsf{Q}, \mathsf{q}_0, \mathsf{R} \rangle$ *où*

- Q *est un ensemble fini d'états de l'automate,*
- $\mathsf{q}_0 \in \mathsf{Q}$ *est l'état* initial *de l'automate,*
- $\mathsf{R} \subseteq \mathsf{Q} \times \mathsf{PRED} \times \mathsf{Q}$ *est un ensemble fini de règles de transition.*

Nous notons BUCHI l'ensemble des automates de Büchi. Un automate de Büchi peut se synchroniser avec une trace σ d'un programme P grâce à une *fonction de synchronisation.*

Définition 3 (Fonction de synchronisation) *Soit* $\mathsf{A} = \langle \mathsf{Q}, \mathsf{q}_0, \mathsf{R} \rangle$ *un automate de Büchi et* $\sigma \in \mathsf{PATH}$ *une trace. La fonction de* synchronisation $\mathsf{sync} : \mathsf{BUCHI} \times \mathsf{PATH} \times \mathbb{N} \to 2^{\mathsf{Q}}$ *est définie récursivement par :*

- $\mathsf{sync}(\mathsf{A}, \sigma, 0) = \{\mathsf{q}_0\}$
- *Pour tout* $i > 0$ *:*

$$\mathsf{sync}(\mathsf{A}, \sigma, i) = \left\{ \mathsf{q}' \;\middle|\; \exists \mathsf{P} \in \mathsf{PRED} \cdot \exists \mathsf{q} \in \mathsf{Q} \cdot \begin{pmatrix} \sigma(i-1) \models \mathtt{P} \wedge \\ \mathsf{q} \in \mathsf{sync}(\mathsf{A}, \sigma, i-1) \wedge \\ \langle \mathsf{q}, \mathtt{P}, \mathsf{q}' \rangle \in \mathsf{R} \end{pmatrix} \right\}$$

On dit qu'un automate de Büchi A **accepte** un programme P si et seulement si, chaque trace σ de P vérifie :

$$\boxed{(C_{Sync}) \quad \forall i \in 0..\mathsf{len}(\sigma).\mathsf{sync}(\mathsf{A}, \sigma, i) \neq \emptyset}$$

4 Génération de conditions suffisantes

En nous basant sur les résultats de [6] montrés dans [7], nous pouvons définir les annotations présentées en figure 5 comme suffisantes pour garantir la synchronisation d'un programme avec un automate de Büchi.

1. Aussi appelées variables ghost

Soit P un programme. Soit $\mathsf{A} = \langle \mathsf{Q}, \mathsf{q}_0, \mathsf{R} \rangle$ un automate de Büchi de sûreté. L'ensemble des clauses d'annotation $\mathsf{CLAUSE}_{\mathsf{Ann}_\mathsf{A}}$ correspondant à A sont :

$$\mathsf{CLAUSE}_{\mathsf{Ann}_\mathsf{A}} = \mathsf{CLAUSE}_{Decl_\mathsf{A}} \cup \mathsf{CLAUSE}_{Trans_\mathsf{A}} \cup \mathsf{CLAUSE}_{Sync_\mathsf{A}}$$

tel que

$$\begin{aligned}
\mathsf{CLAUSE}_{Decl_\mathsf{A}} &= \bigcup_{\mathsf{q}\in\mathsf{Q}} \left\{ \begin{array}{lll} \mathcal{D}(\mathtt{v}_\mathsf{q}, \mathsf{true}) & si & \mathsf{q} = \mathsf{q}_0 \\ \mathcal{D}(\mathtt{v}_\mathsf{q}, \mathsf{false}) & si & \mathsf{q} \neq \mathsf{q}_0 \end{array} \right\} \\
\mathsf{CLAUSE}_{Trans_\mathsf{A}} &= \bigcup_{\mathsf{q}\in\mathsf{Q}} \left\{ \mathcal{A}(\mathtt{v}_\mathsf{q}, \bigvee_{\mathsf{q}'\in\mathsf{Q}\wedge(\mathsf{q}',\mathsf{P},\mathsf{q})\in\mathsf{R}} (\mathsf{P} \wedge \mathtt{v}_{\mathsf{q}'})) \right\} \\
\mathsf{CLAUSE}_{Sync_\mathsf{A}} &= \{ \mathcal{I}(\bigvee_{\mathsf{q}\in\mathsf{Q}} (\mathtt{v}_\mathsf{q})) \}
\end{aligned}$$

FIGURE 5 – Annotations de synchronisation et de description de la propriété

Dans le cadre de la vérification par preuve des annotations générées, nous faisons appel à un générateur d'obligations de preuve (OP) basé sur un calcul de plus faible précondition (WP). Les OP générées sont ensuite déchargées sur un prouveur automatique. Celles associée à la propriété temporelle (donc issues des annotations de la figure 5) ont la forme décrite en figure 6. Intuitivement : si l'automate était synchronisé avant la transition (H_1), alors il doit être synchronisé après la transition (But). Pour prouver cela, nous avons comme hypothèses celles liées au code de synchronisation de l'automate ($H_{2...n}$) et celles liées au corps du programme ($H_{n+1...m}$).

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

FIGURE 6 – Forme générale des obligations de preuve générées

Cette structure rend difficile la preuve automatique, car :

- Les $H_{n+1...m}$ sont difficilement exploitables pour la preuve. En effet, il n'existe pas d'hypothèse faisant le lien entre les états du programme et les états de l'automate.
- L'hypothèse H_1, indiquant que l'on est synchronisé avec au moins un état de l'automate, est souvent trop faible. En effet, s'il existe un état pour lequel aucune transition n'est possible, alors il faut être capable de montrer que l'on ne pouvait pas y être. On retombe alors dans le premier problème.

De ce fait, nous proposons, dans les sections suivantes, de nouveaux algorithmes de génération d'annotations. Ceux-ci vont s'attacher (i) à exploiter les hypothèses sur le contexte du programme, par le biais d'une axiomatisation de l'automate de la propriété, qui fera le lien entre les hypothèses $H_{n+1...m}$ et $H_{2...n}$, (ii) à limiter la disjonction sur les états du But et de l'hypothèse H_1, en excluant des états inatteignables.

5 Génération d'hypothèses supplémentaires

L'objectif de cette section est d'optimiser le processus de génération d'annotations en vue d'obtenir des obligations de preuve qui puissent être déchargées par un prouveur automatique. Pour ce faire, nous proposons :

- en entrée et sortie de chaque opération, d'ajouter dans des pré et post-conditions donnant la liste des états de synchronisation possibles, afin de réduire les disjonctions H_1 et *But*.
- une axiomatisation de l'automate, permettant d'ajouter dans les hypothèses de l'obligation de preuve, un lien logique entre les états du programme et ces de l'automate.

5.1 Ajout de spécification aux opérations

Principe Dans certains cas, pour garantir que la propriété n'est pas violée, il est nécessaire de savoir que le programme ne peut pas être synchronisé avec un état donné. Comme la vérification est effectuée pour chaque opération, indépendamment de son contexte, nous proposons donc d'attacher ce type d'informations à la pré-condition et à la post-condition de chaque opération.

Mise en œuvre Dans un premier temps, nous allons considérer que l'utilisateur aura la charge de préciser, pour chaque opération, les états avec lesquels celle-ci ne peut pas être synchronisée. En section 6, nous proposerons différentes méthodes permettant de générer automatiquement une première approximation de ces annotations.

5.2 Axiomatisation de l'automate

Principe Il n'est souvent pas suffisant d'affiner l'ensemble des états possiblement synchronisés en début et fin d'opération pour vérifier qu'une propriété est respectée par un programme. En effet, ce statut peut dépendre du contexte de l'appel. C'est pourquoi, nous proposons de raffiner la vue que l'on a du système et de considérer non plus seulement le dernier état de synchronisation (*curStates*), mais aussi chacune des transitions pouvant avoir été franchies pour mener à l'un de ces états (*curTrans*). Ici, une transition tr est décrite par un état de départ (*transStart*(tr)), un état d'arrivée (*transStop*(tr)) et une condition de franchissement (*transCond*(tr)). L'ajout de ces transitions permet notamment d'établir que, si aucune des transitions atteignant un état E donné n'a sa condition de franchissement vérifiée, alors le système ne peut pas être synchronisé avec E.

Mise en œuvre Dans un soucis de simplification, la spécification de chaque opération sera augmentée (et non remplacée) par l'ensemble des transitions

ayant pu être franchies. Enfin, pour simplifier l'expression des invariants, nous mémorisons également le précédent ensemble des états synchronisés avec le système ($curStates_{Old}$).

```
/*@ 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 */) ∧
  @   ...  */
```

FIGURE 7 – Axiomatisation de l'automate de la propriété vérifiée

L'automate étant constant, il est décrit sous la forme de constantes logiques (d'axiomes), comme montré en figure 7. Pour exploiter cet automate, nous introduisons également différents invariants indiquant, par exemple, que l'on ne peut pas être synchronisé avec un état dans lequel le système n'a pas pu aller par le franchissement d'une transition (figure 8).

```
/*@ global invariant Non-atteignabilité_1 :
  @   ∀st; 0 ≤ st < NbStates ∧
  @         ( ∀tr; 0 ≤ tr < NbTrans
  @               ⇒ curTrans[tr] = 0 ∨ transStop(tr) ≠ st ∨
  @                 ¬transCond(tr) ∨ curStates_Old[transStart(tr)] = 0 )
  @     ⇒ curStates[st] = 0;   */
```

Intuitivement : *pour qu'un état st ne soit pas synchronisé, il suffit que, pour toute transition tr : soit tr n'est pas mémorisée comme franchie, soit elle ne mène pas à st, soit sa condition de franchissement est fausse, soit le système n'était pas synchronisé avec son état de départ à l'instant d'avant.*

FIGURE 8 – Exemple d'invariant de non-atteignabilité généré

Le code issu de la clause CLAUSE_{Sync_A} (figure 5) et permettant de synchroniser le programme avec l'automate est donc adapté par rapport aux travaux initiaux [6] pour prendre en compte les nouvelles données introduites. Un exemple est présenté en figure 9.

Avec la mémorisation des transitions et l'introduction de divers invariants permettant de lier et d'exploiter les différentes informations, le nombre d'obligations de preuve augmente très rapidement avec le nombre d'états ou de transitions. Dans la section suivante, nous proposons différentes analyses statiques permettant de prévoir des cas impossibles, afin de réduire le nombre et la complexité des OP générées.

```
/* Mise à jour des variables de statut */
Oper = op_commit /* Nom de l'opération */;
St = Call /* Statut de l'opération (Appel ou retour)*/;

/* Mise à jour des anciens états */
for (st = 0 ; st < NbStates ; st++) {
   curStates_Old[st] = curStates[st];
   curStates[st] = 0
}

/* Mise à jour des transitions et des états */
for (tr = 0 ; tr < NbTrans ; tr++) {
   tmp = curStates_Old[transStart(tr)] ∧ transCond;
   curTrans[tr] = tmp;
   curStates[transStop(tr)] = curStates[transStop(tr)] ∨ tmp
}
```

FIGURE 9 – Code de synchronisation généré

6 Limitation de l'explosion combinatoire

Dans cette section, tout l'enjeu consiste à réduire statiquement les espaces d'états et de transitions acceptés en entrée et sortie de chaque opération.

Notons que la correction du système ne peut pas être altérée par ces simplifications. En effet, si la spécification est trop simplifiée et qu'un état atteignable en est retiré, alors le code qui est généré pour calculer la synchronisation avec l'automate mènera dans un état interdit et les obligations preuve ne pourrons pas être vérifiées.

Pour spécifier les opérations, nous proposons une approche en 2 temps. Dans un premier temps, nous spécifions les opérations à partir de la propriété. Cette première approche grossière est ensuite raffinée par propagation des contraintes au travers du flot de contrôle, par interprétation abstraite.

6.1 Sur-approximation depuis la propriété

Les gardes des transitions de l'automate sont décrites en termes d'expressions, portant sur les variables du programme, et de prédicats, décrivant l'appel ou le retour d'une opération. En s'abstrayant des expressions, il est possible d'exploiter les prédicats pour prédire statiquement qu'une transition ne peut pas être franchie par l'appel ou le retour d'une opération donnée et pour pouvoir retirer cette transition de la spécification de l'opération.

Par exemple, l'appel de l'opération *commit* ne peut franchir aucune des transitions de la propriété décrite en figure 3. Sa pré-condition se réduit donc au franchissement de la transition allant de l'état 0 vers l'état 1.

Notons $Pré(Op)$ la pré-condition d'une opération et $Post(Op)$ sa post-

condition, telles que $Pré(Op).trans$ est l'ensemble des transitions qui peuvent être franchies par l'appel d'Op et que $Pré(Op).état$ est l'ensemble des états de synchronisation autorisés. Il est alors possible, à partir de l'automate de la propriété, de calculer une sur-approximation de la spécification de chaque opération comme suit :

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

Où la notation $F[E]$ est l'image de l'ensemble E par la fonction F.

Cette première approche permet de retirer des transitions directement interdites. Enfin, la spécification de l'opération *main* peut être raffinée, car étant l'unique point d'entrée du programme étudié, ses états d'entrée sont nécessairement des états initiaux de la propriété. Dans les sections suivantes, nous proposons de propager ces contraintes à travers le flot de contrôle du programme.

6.2 Propagation statique des contraintes

À ce stade du traitement, chaque opération a une pré et une post-condition autorisant tous les états et toutes les transitions sauf ceux qui sont explicitement interdits par la propriété. Nous proposons maintenant de propager ces contraintes en réalisant une interprétation abstraite avant-arrière [2], où l'on ne considère que les états et les transitions de la propriété (on s'abstrait des expressions relatives aux variables du système). Une première passe (en avant) permet, pour chaque opération, de propager sa pré-condition à travers son corps, jusqu'à restreindre sa post-condition par rapport à l'ensemble des états atteignables. De la même manière, une deuxième passe (en arrière) permet de restreindre la pré-condition à ceux de ses états qui ne mènent pas nécessairement dans une post-condition interdite.

Cet algorithme est défini par induction sur l'ensemble des instructions. Étant donnés une pré-condition P et une séquence d'opérations L, la figure 10 illustre la propagation de la pré-condition $Fwd(P, L)$ de manière intuitive sur certaines instructions clé.

Lorsqu'un appel d'opération est rencontré durant la passe en avant, c'est la post-condition de l'opération qui devient la pré-condition propagée, tandis que durant la passe en arrière, c'est sa pré-condition qui devient la post-condition propagée.

De plus, le franchissement des transitions est effectué lors de chaque ap-

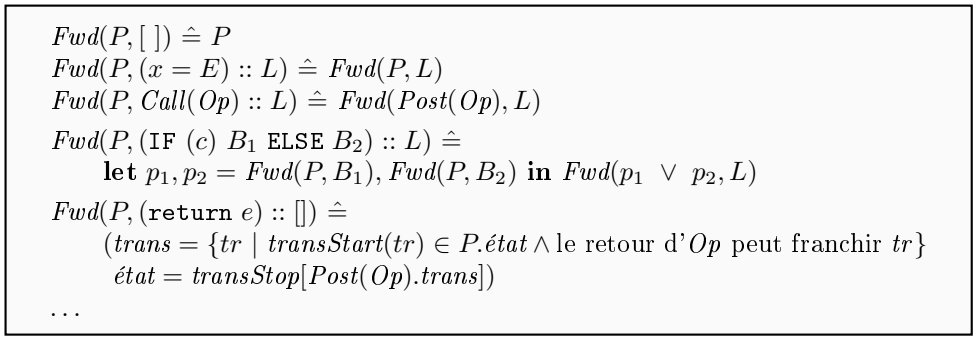

FIGURE 10 – Exemple de propagation de pré-condition sur quelques instructions

pel et de chaque retour d'opération. Ce code n'étant pas encore généré, il est donc nécessaire de le simuler en déterminant l'ensemble des transitions qui peuvent être franchies, comme l'illustre le cas du retour. Dans l'algorithme de propagation de la post-condition, l'approche est similaire mais c'est l'instruction d'appel qui nécessite de calculer le franchissement de transitions. La post-condition (resp. pré) d'une opération est alors l'intersection entre sa post-condition (resp. pré) issue de la propriété et celle propagée par *Fwd* (resp. *Backward*) :

$$\begin{array}{lcll} Post(Op) & \leftarrow & Post(Op) & \cap \quad Fwd(Pré(Op), body(Op)) \\ Pré(Op) & \leftarrow & Pré(Op) & \cap \quad Backward(Post(Op), body(Op)) \end{array}$$

Dans [11], ce sont les variables d'annotation qui sont propagées et la propagation est effectuée dans le sens contraire. Par exemple, la pré-condition d'une opération o y calculée comme l'union des pré-conditions des instructions de o, diminuée des prédicats portant sur des variables modifiées avant leur apparition. Notre approche est donc beaucoup plus fine, grâce au fait que l'on ne traite pas des prédicats logiques mais des ensembles d'états et de transitions. Nous ne générons les prédicats associés qu'une fois toutes les analyses effectuées.

Restriction aux cas d'utilisation Durant cette analyse, l'intégralité des appels d'opération est observé. Il est donc possible de recenser l'ensemble des cas d'utilisation de chaque opération. Ainsi, lorsqu'une opération n'est jamais appelée depuis certains de ses états d'entrée autorisés, ces derniers sont supprimés de la liste, simplifiant d'autant les obligations de preuve générées. Le même traitement est effectué pour les post-conditions.

Spécification des boucles Propager une pré ou une post-condition revient à associer une spécification à chaque (bloc d')instruction(s). En particulier, pour chaque boucle, il est nécessaire de définir ses pré et post-conditions

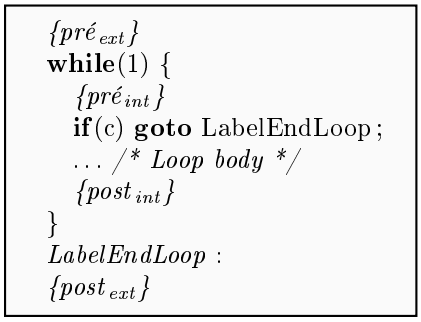

FIGURE 11 – Forme d'une boucle (dans l'outil Frama-C) avec annotations

et de lui générer un invariant en terme des états et transitions de la propriété.

Cette étude étant réalisée dans le cadre de l'outil Frama-C, nous pouvons prendre en compte les pré-traitements qu'il effectue. Ainsi, toute boucle est sous la forme décrite en figure 11. Étant données ses deux pré-conditions (de la boucle et de son corps) et ses deux post-conditions, l'invariant d'une boucle peut s'exprimer de la manière suivante :

$$//@\ \textbf{Loop invariant}\ \mathrm{i} : (\mathit{Init} \wedge \mathit{Pré}_{ext}) \vee (\neg \mathit{Init} \wedge \mathit{Post}_{int})$$

où *Init* est une variable fraîche, identifiant la première itération.

Le calcul des assertions, correspond à un calcul de point fixe utilisant *Fwd* et *Backward* à partir soit de $\mathit{Pré}_{ext}$, pour la passe en avant, soit de Post_{ext}, pour la passe en arrière. Au final, l'introduction de la disjonctionpermet d'affiner l'invariant. Notons quand dans notre cas particuler où l'on connait la pré et la post-condition de la boucle, la génération d'un invariant disjonctif ne se heurte pas aux coûts théoriques de la génération d'invariants disjonctifs [3].

Raffinement des post-conditions Enfin, nous proposons de raffiner la représentation manipulée des post-conditions pour augmenter la précision des spécifications. Pour ce faire, nous considérons les états et transitions autorisés en sortie d'une opération en terme des états d'entrée de celle-ci. Typiquement, on obtient des post-conditions décomposées en comportements de la forme :

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

Comme l'ensemble possible des états d'entrée est connu de l'appelant, cela lui permet de ne considérer que les états de sortie associés. Dans la

pratique, cette méthode de description permet de simplifier les spécifications calculées, en particulier dans les structures de contrôles pouvant avoir plusieurs sources (boucles, cibles de goto, sortie de conditionnelles, *etc*).

7 Développements réalisés

Les travaux décrits ici ont été implantés dans le plug-in « Aoraï », qui s'intègre dans la plate-forme Frama-C [2]. Développée par le CEA-List et l'INRIA-Saclay – Île-de-France, Frama-C est une suite open-source d'outils dédiés à l'analyse de programmes C annotés (avec le langage ACSL). La figure 12 résume la démarche générale de l'utilisation du Plug-in Aoraï et de ses interactions avec les outils extérieurs.

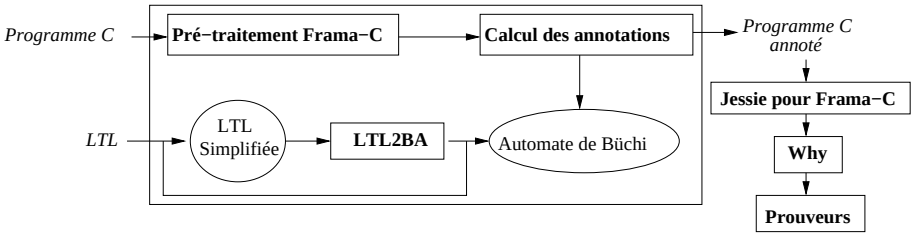

FIGURE 12 – Vue générale de fonction du plug-in

L'outil LTL2BA [5], développé au LSV [3], est utilisé pour construire un automate de büchi représentant une propriété LTL. Certaines restrictions de langage nous obligent à retirer les expressions de la formule LTL initiale et de les replacer ensuite dans l'automate produit.

Le plug-in Jessie [10] de Frama-C est ensuite utilisé pour l'analyse déductive du programme. En interne, il utilise l'outil Why [4] pour générer les obligations de preuve et les décharger sur différents prouveurs tels que Simplify [4] ou Alt-Ergo [1].

Application au programme fil rouge La vérification du programme fil rouge par rapport à la propriété présentée en figure 3 génère 10174 obligations de preuve en utilisant le calcul de WP classique, contre seulement 296 en utilisant la méthode de calcul efficace du WP proposée par K. Rustan M. Leino [9] (implantée dans Why sous le nom de *Fast WP*). Cette méthode consiste à ne pas systématiquement séparer les OPs liées à un même branchement dans le flot de contrôle du programme. Le nombre d'obligations de preuve peut donc être réduit, mais leur complexité est augmentée.

2. `http://www.frama-c.cea.fr/`
3. `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`
4. `http://research.compaq.com/SRC/esc/Simplify.html`

Toutefois, l'approche de WP efficace est bien adaptée à notre cas de figure, puisque les OPs que nous générons sont souvent très simples grâce aux multiples instanciations réalisées. De plus, comme un grand nombre des obligations de preuve générées sont similaires (ne diffèrent ni par leur but, ni par leurs hypothèses pertinentes[5]), alors l'utilisation du WP efficace permet de limiter la duplication de ces obligations de preuve. En effet, comme la granularité des propriétés vérifiée est l'appel et le retour d'opérations, le code se situant entre deux appels génère principalement du *« bruit »* dans les obligations de preuve.

Ainsi, sur les 296 OPs générées, 289 sont vérifiées automatiquement. Les 7 restantes sont actuellement fausses. En effet, les obligations de preuve sont générées par le plug-in Jessie qui est en cours de développement. Ainsi, il ne prend pas encore en compte la valeur initiale des tableaux dans les OPs liées à la vérification des invariants lors de l'initialisation.[6]

8 Bilan et perspectives

Résultats obtenus Nous avons proposé une extension de travaux existants [6] dans le domaine de la vérification de propriétés LTL, en axant notre proposition sur l'aspect automatique de la résolution des obligations de preuve générées.

Dans un premier temps (section 5), nous avons renforcer la spécification générée dans le but de lier plus fortement le programme et l'automate, simplifiant d'autant les raisonnements possibles lors de la résolution d'obligations de preuve. Dans une deuxième partie (section 6), nous avons voulu diminuer le nombre et la complexité des obligations de preuve générées en restreignant l'espace d'états de la spécification. Pour ce faire, nous propageons statiquement les contraintes liées soit à la propriété vérifiée, soit au flot de contrôle du programme. Toutes ces propositions ont été validées par une implantation de 7400 lignes d'Ocaml au sein du plug-in Aoraï de Frama-C.

Comparaison avec des travaux existants La propagation de propriétés a été abordée dans différents travaux, dont le plus proche est celui réalisé dans le cas de l'outil Jack [11], où les auteurs s'intéressent à la propagation d'annotations JML représentant des propriétés de haut niveau. Leur approche se distingue de la notre par la dissociation des processus. En effet, ils proposent une méthode itérative, consistant à générer des annotations, puis à les propager, nécessitant ainsi la propagation de toutes les annotations logiques. À l'inverse, l'approche que nous présentons dans cet article consiste à propager les contraintes liées à la propriété de haut niveau, avant

5. Hypothèses nécessaires à la validation d'une propriété.
6. Normalement, ces limitations devraient être corrigées d'ici fin 2008.

de générer les annotations associées, ce qui permet de maîtriser la sémantique des différentes annotations lors de leur propagation.

Travaux futurs Actuellement, deux aspects semblent importants : la précision des annotations générées et les hypothèses spécifiques à la prise en compte de la vivacité.

D'un point de vue génération d'OP, il serait possible d'affiner encore la propagation des annotations générées en remplaçant l'interprétation par des calculs de WP ou de SP. L'objectif serait alors de pouvoir caractériser, si nécessaire, les états possibles d'entrée ou sortie d'une opération en terme des valeurs des variables du programme.

Enfin, la vérification de la composante vivacité des propriétés a été proposée dans les travaux originaux et nos apports ne peuvent pas nuire à la vérification des OP générées en ce sens. Nous prévoyons toutefois d'effectuer le même type d'optimisations sur ces obligations de preuve, afin de favoriser l'automatisme du mécanisme de vérification. En particulier, il est possible d'exploiter les mécanismes de variants présents dans le langage d'annotation ACSL.

Références

[1] S. Conchon, E. Contejean, and J. Kanig. CC(X) : Efficiently Combining Equality and Solvable Theories without Canonizers. In *5th International Workshop on Satisfiability Modulo*, Berlin, Germany, July 2007.

[2] P. Cousot and R. Cousot. Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

[3] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282, 1979.

[4] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[5] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV'01*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[6] J. Groslambert. Verification of LTL on B event systems. In *B'2007, 7th International B Conference*, LNCS, Besancon, France, January 2007. Springer-Verlag.

[7] Julien Groslambert. *Vérification de propriétés temporelles par génération d'annotations*. PhD thesis, Université de Franche-Comté, Septembre 2007.

[8] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3) :185–208, 2005.

[9] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6) :281–288, 2005.

[10] Y. Moy. Sufficient preconditions for modular assertion checking. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, LNCS. SV, jan 2008.

[11] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J-L. Lanet. Enforcing high-level security properties for applets. In J-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. El Kalam, editors, *CARDIS*, pages 1–16. Kluwer, 2004.

[12] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *AMAST'02*, number 2422 in LNCS, pages 334–348. Springer, 2002.