Au temps en emporte le C

Steven De Oliveira, Virgile Prevosto, Sébastien Bardin

CEA, LIST, Laboratoire de Sûreté des Logiciels
91191 Gif-sur-Yvette Cedex
`prenom.nom@cea.fr`

Résumé

Le (bon) séquencement des événements au fil du temps est un sujet important pour l'analyse de programmes, par exemple dans le cadre de protocoles d'échange d'information ou de systèmes embarqués. De nombreux travaux s'appuient sur la logique temporelle linéaire (LTL) pour décrire formellement le comportement attendu d'un programme, sous la forme d'une succession d'actions distinctes. Implanté au sein de Frama-C, plate-forme d'analyse de code source en langage C, le greffon Aoraï permet la génération de contrats de fonctions équivalents à la satisfaisabilité d'une formule LTL pour un programme C donné. Cependant le greffon montre certaines limitations. Cet article présente *CaFE*, un nouveau greffon Frama-C qui pallie ces manques. Plus précisément, *CaFE* vérifie automatiquement de manière approchée (mais correcte) qu'un programme C vérifie une formule logique *CaRet* donnée, où *CaRet* est une extension de LTL autorisant en particulier des raisonnements explicites sur la pile d'appel du programme.

1. Introduction

Frama-C [10] est un atelier libre et extensible à vocation industrielle constitué d'une constellation de greffons indépendants ou interagissant entre eux afin de vérifier différents types de propriétés sur des programmes écrits en C. Les prédicats permettant de représenter formellement ces propriétés sont exprimés en ACSL [5], un langage de spécification utilisé par la plupart des greffons de Frama-C. ACSL est un langage basé principalement sur la notion de contrats de fonctions [17]. Un contrat spécifie formellement pour chaque fonction C ce qu'elle requiert de ses appelants (ses pré-conditions), et ce qu'elle garantit en retour à la fin de son exécution (ses post-conditions).

Certaines classes de propriétés sont assez difficiles à exprimer en ACSL, en particulier lorsqu'on s'intéresse à l'évolution de l'état du programme à travers plusieurs appels de fonctions. En revanche, les logiques temporelles [21] sont des outils appropriés pour la spécification de telles propriétés. En outre, elles permettent l'étude de traces d'exécution infinies, ce qui est là encore difficile à réaliser en ACSL. En particulier, LTL [20] (Linear Temporal Logic, Logique Temporelle Linéaire) interprète le temps comme un chemin linéaire et permet d'exprimer des propriétés sur l'ensemble de ce chemin, comme «à l'instant suivant, φ sera vraie» ou «φ est vraie tant que ψ ne l'est pas». Parmi les propriétés exprimables en LTL, on distingue deux grandes catégories. Les propriétés *de sûreté* expriment le fait que quelque chose ne doit pas arriver, tandis que les propriétés de *vivacité* expriment que quelque chose doit finir par arriver.

Aoraï [22] est un greffon de Frama-C permettant la vérification de formules LTL sur un programme C. Le greffon fait la passerelle entre programme et automate [13] et permet une étude du système en des points clefs prédéfinis (en début et en fin de fonction). Cependant, Aoraï montre quelques faiblesses dans son mécanisme de spécification. Tout d'abord, LTL ne contient pas d'opérateurs permettant de parler simplement de la pile d'appels du programme. Ensuite, Aoraï n'a pas la possibilité d'analyser

des traces d'exécution infinies et se restreint donc à des propriétés de sûreté. Finalement, la granularité d'observation est fixée, ce qui peut être gênant.

Mettons-nous par exemple dans le domaine de la programmation concurrente dans lequel plusieurs propriétés spécifiques sont attendues, notamment de sûreté et de vivacité. Lorsque plusieurs processus travaillent sur une même ressource, on doit pouvoir gérer les priorités, les droits d'accès, etc... On utilise en général un *verrou* exprimant la disponibilité d'une ressource au cours du temps. Nous souhaitons pouvoir exprimer sur de tels systèmes les garanties suivantes :

1. Sur la fonction principale, la donnée est libre (garantie de sûreté) ;
2. Si une fonction a été appelée, alors à son appel, la ressource était libre (garantie contextuelle) ;
3. Toute fonction doit libérer la ressource avant de terminer (garantie de vivacité).

Il est alors nécessaire d'avoir un mécanisme d'étude des changements de l'état du verrou le long d'une exécution eventuellement infinie (sûreté) et le long de la pile d'appel (contexte). De plus, il serait intéressant de pouvoir travailler uniquement sur le contenu d'une seule et même fonction sans observer ce qui est effectué en dehors (sûreté et vivacité).

Pour aller au-delà des possibilités actuelles d'Aoraï et pouvoir répondre à ces besoins, nous avons développé le nouveau greffon *CaFE* (*CaRet Frama-C Extension*) :

- le greffon est basé sur la logique *CaRet* [2], une extension de LTL permettant de manipuler et d'étudier directement la pile d'appel ;
- le greffon prend en compte aussi bien la sûreté que la vivacité ;
- la granularité d'observation est modulable et peut recouvrir l'ensemble des instructions.

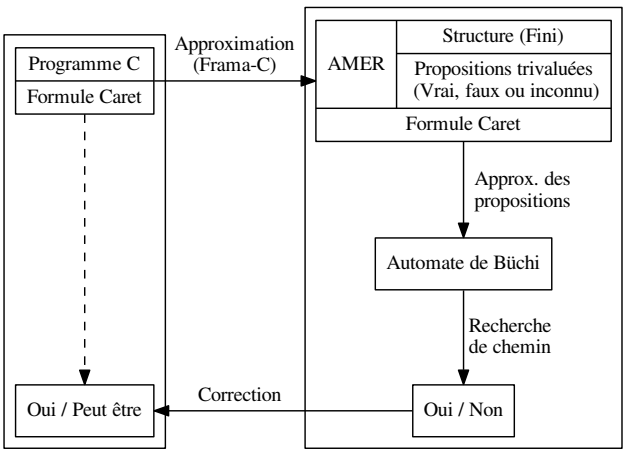


FIGURE 1 – Fonctionnement de *CaFE*

Le fonctionnement de *CaFE* est résumé dans la figure 1. Pour montrer qu'un programme P vérifie une propriété *CaRet* φ donnée, *CaFE* commence par abstraire P sous forme d'Abstraction de Machine à États Récursif (*AMER*), c'est à dire sous forme de Machine à états Récursifs (*MER*) [1] (équivalente à un automate à pile) dans laquelle les états sont étiquetés par des propositions atomiques tri-valuées. Cette approximation finie se base sur la sémantique collectrice [19] du programme, projetée sur les valuations des prédicats atomiques observés. Nous l'obtenons actuellement par appel au greffon d'interprétation abstraite Value [8, 10] de Frama-C. Nous adaptons ensuite l'algorithme classique de

model-checking de *CaRet* sur *MER* [2] aux *AMER*. Notre procédure est correcte, au sens où si la réponse est positive, alors P satisfait bien φ. La procédure n'est pas complète, puisqu'elle comporte deux phases d'approximations : le passage d'un programme C ($\approx$ *MER* + données infinies) à une *AMER* fini, et la résolution approchée de *CaRet* sur des *AMER*.

Cet article présente l'implantation de l'algorithme de vérification de propriétés *CaRet* adapté à son utilisation dans le cadre de Frama-C. La section 2 résume les travaux existants autour de LTL et du model-checking de programmes C. La logique *CaRet* est présentée dans la section 3 avec les modifications que nous y avons apportées. La section 4 décrit l'algorithme de transformation d'une formule *CaRet* et d'un automate à états récursifs en un second automate. La condition d'acceptation de ce dernier est équivalente à la satisfaction de la formule initiale par les parcours du premier automate considéré. Cette description présente les optimisations que nous apportons à l'algorithme afin de réduire le nombre de tests nécessaire à la génération du second automate. Enfin, la section 5 détaille l'implantation de cet algorithme dans le cadre de Frama-C et montre l'utilisation de *CaFE* sur un exemple que nous développerons tout au long de cet article.

2. Travaux connexes

2.1. Logique Temporelle Linéaire

Les formules LTL expriment des propriétés sur des suites (infinies) d'événements.

Définition 1. *Soit AP un ensemble de propositions. Nous définissons $\varphi \in LTL$ de la manière suivante :*

$$\varphi ::= \top \mid p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid \varphi U \varphi$$

La relation de satisfaction $\vDash$ d'une formule LTL φ par une suite $\pi = (\pi_i)_{i\in\mathbb{N}}$ sur 2^{AP} est définie par :

$$\begin{aligned}
(\pi, i) &\vDash \top \\
(\pi, i) &\vDash p \quad \Leftrightarrow \quad p \in \pi_i \\
(\pi, i) &\vDash \neg\varphi \quad \Leftrightarrow \quad (\pi, i) \nvDash \varphi \\
(\pi, i) &\vDash \varphi \wedge \psi \quad \Leftrightarrow \quad (\pi, i) \vDash \varphi \text{ et } (\pi, i) \vDash \psi \\
(\pi, i) &\vDash X\varphi \quad \Leftrightarrow \quad (\pi, i+1) \vDash \varphi \\
(\pi, i) &\vDash \varphi U \psi \quad \Leftrightarrow \quad \exists n \geqslant i, (\pi, n) \vDash \psi \text{ et } \forall i \leqslant j < n, (\pi, j) \vDash \varphi
\end{aligned}$$

On peut en outre définir deux modalités supplémentaires $F\varphi = \top\ U\ \varphi$ (Finalement, φ sera vérifiée) et $G\varphi = \neg(F(\neg\varphi))$ (φ est vérifiée partout).

2.2. Aoraï

Aoraï [22], le greffon de Frama-C permettant la spécification de programmes en LTL, avec des propositions atomiques formées par un sous-ensemble d'ACSL, se base sur une transformation de la formule LTL en un automate de Büchi [12]. Étant donné un programme C censé respecter la propriété LTL, l'outil va alors générer des marqueurs sous la forme de fonctions en début et fin de chacune des fonctions du programme initial. Ces fonctions gèrent l'évolution de l'automate, de façon à ce que le programme et l'automate s'exécutent en "parallèle". De plus, toutes les fonctions se voient dotées d'un contrat ACSL. La vérification de l'ensemble de ces contrats est équivalente à la condition d'acceptation de l'automate [13], c'est à dire à vérifier que le programme est bien conforme à la propriété LTL.

Cette approche du temps en fait un découpage efficace mais large : les seuls événements étudiés sont les appels et les retours de fonction. Il est donc possible qu'une propriété qu'on souhaite maintenir le long de l'exécution devienne fausse entre deux marqueurs, puis redevienne vraie avant le suivant.

Prenons l'exemple de la propriété suivante : "La variable x doit toujours être à 0." s'exprimant en LTL : "G (x == 0)". Aoraï ne fera ses tests qu'aux marqueurs qu'il a défini, ainsi la propriété que vérifiera Aoraï sera : "La variable x doit toujours être à 0 au début et à la fin de chaque fonction.". De plus, la génération de spécifications à la fin de chaque fonction impose au module principal de pouvoir y accéder, sans quoi ils ne seraient pas vérifiables. Ainsi, tous les modules doivent terminer, obligeant l'outil à limiter son analyse aux traces d'exécutions finies. Or, LTL s'intéresse à des traces potentiellement infinies. En réalité, Aoraï se restreint à un sous-ensemble de LTL.

2.3. Model-checking

Il existe déjà un certain nombre d'outils de vérification de formules de logique temporelle sur des automates finis tels que SMV[9] et SPIN [14] traitant ce problème de manière efficace. Cependant, des outils basés sur les automates ne permettent pas de travailler directement sur des programmes. Puisqu'un programme possède une structure potentiellement infinie (données manipulées de taille arbitraire), une phase d'approximation est nécessaire. Quelques automates à structure infinie ont été étudiés afin de minimiser cette approximation, comme par exemple les automates à pile [11] dont la complexité en temps du model-checking dans le pire des cas est cubique en la taille de l'automate.

Différents outils basés sur des techniques de model-checking ont cependant été proposés pour l'analyse de code [15]. En particulier, BLAST et son successeur CPAchecker [7], LLBMC [16], ou SLAM [3]. Cependant, ils se restreignent à l'expression de propriétés de sûreté [4].

3. *CaRet*

La logique *CaRet* [2] est une extension de LTL, dont l'intérêt principal est de permettre, grâce à des modalités adaptées, l'étude de la trace d'exécution d'un programme dans son intégralité et celle d'un module (i.e. une fonction) en particulier en utilisant une pile d'appel comme une mémoire des propriétés vraies à l'entrée d'un module. Nous présentons cette logique dans le cadre des *abstractions de machines à états récursifs*, généralisation des *machines à états récursifs* [1] utilisées pour *CaRet*.

3.1. Abstraction des machines à états récursifs

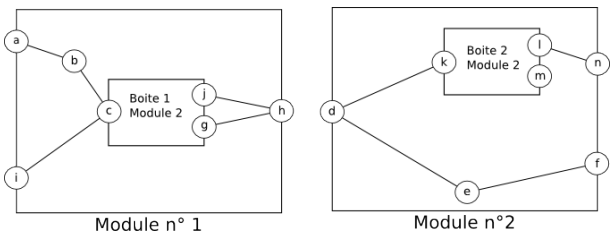


FIGURE 2 – Exemple d'*AMER*

Les *MER* (Recursive state machine [1] ou Hierarchical state machine [6]) sont des machines finies équivalentes à des automates à pile. Elles ont l'avantage de représenter correctement et intuitivement le graphe de flot de contrôle inter-procédural d'un programme.

Ce sont des automates contenant des états particuliers appelés *boîtes*. Lorsqu'une exécution de la machine à états récursifs (*MER*) passe par une boîte, elle entre, selon sa transition d'entrée, dans un autre module de la *MER* et y continue son exécution. Lorsqu'elle sort de cette dernière, elle revient à la sortie de la boîte et repart. La figure 2 présente un exemple de *MER* comportant deux modules. Le module 1 appelle le module 2, quand il arrive en `c`. Lorsque l'exécution du module 2 se termine, le module 1 reprendra alors son exécution en `g` ou `j`. De même, le module 2 peut s'appeler récursivement. Plus généralement, on définit les *AMER* de la manière suivante.

Définition 2. *Soit AP un ensemble de propositions. Une abstraction de machine à états récursifs R sur AP est un quadruplet* $(M, \{R_m\}_{m\in M}, \eta, init)$*, où M est un ensemble fini d'identifiants et pour chaque* $m \in M$*, il existe un module* $R_m = (N_m, B_m, Y_m, En_m, Ex_m, Calls_m, Retns_m, \delta_m)$ *tel que :*

- N_m *est un ensemble fini de nœuds.*
- B_m *est un ensemble fini de boîtes.*
- $Y_m : B_m \to M$ *est une application associant à chaque boîte la référence d'un module.*
- En_m *et* Ex_m *sont deux sous ensembles non vides de* N_m*, représentant respectivement les nœuds d'entrée et de sortie d'un module.*
- $Calls_m = \{(b, e) | b \in B_m,\ e \in En_{Y_m(b)}\}$ *et* $Retns_m = \{(b, x) | b \in B_m,\ x \in Ex_{Y_m(b)}\}$
- $\delta_m : N_m \cup Retns_m \to 2^{N_m \cup Calls_m}$ *est une fonction de transition.*
  Par la suite, on notera δ *et* Y *le prolongement des fonctions* δ_m *et* Y_m *ainsi que* N*,* B*,* En*,* Ex*,* $Calls$ *et* $Retns$ *l'union des ensembles* N_m*,* B_m*,* En_m*,* Ex_m*,* $Calls_m$ *et* $Retns_m$ *pour chaque* $m \in M$*.*
- $\eta : (N \cup Calls \cup Retns) \times AP \to \{\top, \bot, \star\}$ *est une fonction associe à un noeud u et à une propriété p,* $\top$ *si p est vérifiée en u,* $\bot$ *si elle ne l'est pas et* $\star$ *si on ne sait pas.*
- $init \subseteq N$ *est l'ensemble des états initiaux de R.*

Une *AMER* dont la fonction η ne renvoie jamais $\star$ est une *MER*.

3.2. Mots imbriqués

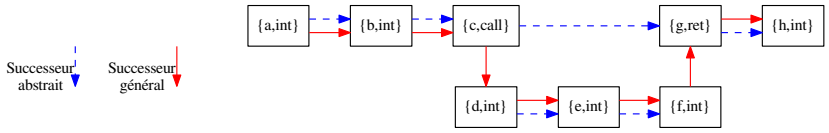
Soit Γ un ensemble de symboles, $\Gamma^* = \Gamma \times \{\texttt{call}, \texttt{ret}, \texttt{int}\}$ l'alphabet étendu de Γ. Les indicateurs `call` et `ret` indiquent l'entrée et la sortie d'un module, et `int` les opérations internes à un module. On peut considérer soit la suite de tous les événements (la trace générale) soit la suite d'événements comprise dans un même module (la trace abstraite, dans laquelle on relie directement chaque `call` au `ret` qui lui est associé). Pour un mot $\gamma = \gamma_0\gamma_1...\gamma_n$, on définit deux fonctions partielles C_γ et R_γ. $C_\gamma(i)$ est l'indice du précédent `call` de γ avant γ_i s'il existe et $\bot$ sinon. De même, $R_\gamma(i) = \bot$ si γ_i est un `ret`, et sinon le prochain `ret` de γ à partir γ_i. On peut alors définir différents successeurs :

— $succ^g_\gamma$: le successeur usuel, c'est à dire $succ^g_\gamma(i) = i + 1$
— $succ^a_\gamma$: le successeur abstrait, qui pointe vers le prochain successeur local c'est à dire $succ^a_\gamma(i) = i + 1$ si γ_i est un `int`, $R_\gamma(i)$ sinon.
— $succ^-_\gamma(i) = C_\gamma(i)$

Lorsqu'il n'y aura pas d'ambiguïté sur γ, on notera $succ^b_\gamma = succ^b$ pour $b \in \{a, g, -\}$.

Par exemple, considérons le mot $\gamma : (a, \texttt{int}), (b, \texttt{int}), (c, \texttt{call}), (d, \texttt{int}), (e, \texttt{int}), (f, \texttt{int}), (g, \texttt{ret}), (h, \texttt{int})$ présenté figure 3. Il représente une exécution possible de l'*AMER* de la figure 2. L'appel au module 2 depuis le module 1 se fait au niveau du nœud c, et le retour correspondant est au nœud g. Les fonctions $succ^b_\gamma$ sont les suivantes :

nœud	a	b	c	d	e	f	g	h
$succ^g$	b	c	d	e	f	g	h	$\bot$
$succ^a$	b	c	g	e	f	$\bot$	h	$\bot$
$succ^-$	$\bot$	$\bot$	$\bot$	c	c	c	$\bot$	$\bot$



a, b, c ,g et h sont au même niveau, i.e. dans le même module

FIGURE 3 – Exemple de chemin d'exécution sur une *AMER*

Par la suite, on parlera d'un mot représentant une trace d'exécution d'une *AMER* A comme d'un *chemin* sur A.

3.3. Définition de *CaRet* sur une *AMER*

Nous allons maintenant définir la sémantique de *CaRet* sur une *AMER* R, dans laquelle un choix est nécessaire. En effet, η peut répondre $\star$, mais nous souhaitons garder une logique bi-valuée. Afin de pouvoir présenter une réponse correcte, il faut interpréter $\star$ en fonction des propriétés qu'on souhaite avoir. Nous proposons ici de faire une sur-approximation des formules acceptées, c'est à dire de considérer que si $\eta(e,p) = \star$, alors p comme $\neg p$ peuvent être vérifiées.

Définition 3. *Soit AP un ensemble de propositions. Nous définissons la logique* CaRet *de la manière suivante :*

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X^g\varphi \mid X^a\varphi \mid X^-\varphi \mid \varphi U^g \varphi \mid \varphi U^a \varphi \mid \varphi U^- \varphi$$

Soient maintenant R une AMER sur AP, N l'ensemble des noeuds de R. $\pi = (\pi_i)_{i\in\mathbb{N}}$ un mot sur $\Gamma^* = N \times \{\texttt{call}, \texttt{ret}, \texttt{int}\}$, et φ une formule CaRet*. On définit la relation de satisfaction de φ par π, notée $(\pi, i) \vDash \varphi$, de la manière suivante, avec $(b \in \{a, g, -\})$ et $i \in \mathbb{N} \cup \{\bot\}$.*

$$\begin{aligned}
(\pi,i) \vDash p &\Leftrightarrow i \neq \bot \text{ et } \eta(\pi_i, p) \in \{\top, \star\} \\
(\pi,i) \vDash \neg p &\Leftrightarrow i \neq \bot \text{ et } \eta(\pi_i, p) \in \{\bot, \star\} \\
(\pi,i) \vDash \varphi \wedge \psi &\Leftrightarrow (\pi,i) \vDash \varphi \text{ et } (\pi,i) \vDash \psi \\
(\pi,i) \vDash \neg(\varphi \wedge \psi) &\Leftrightarrow (\pi,i) \vDash \neg\varphi \text{ ou } (\pi,i) \vDash \neg\psi \\
(\pi,i) \vDash \varphi \vee \psi &\Leftrightarrow (\pi,i) \vDash \varphi \text{ ou } (\pi,i) \vDash \psi \\
(\pi,i) \vDash \neg(\varphi \vee \psi) &\Leftrightarrow (\pi,i) \vDash \neg\varphi \text{ et } (\pi,i) \vDash \neg\psi \\
(\pi,i) \vDash X^b\varphi &\Leftrightarrow (\pi, succ^b(i)) \vDash \varphi \\
(\pi,i) \vDash \neg X^b\varphi &\Leftrightarrow succ^b(i) = \bot \text{ ou } (\pi, succ^b(i)) \vDash \neg\varphi \\
(\pi,i) \vDash \varphi U^b \psi &\Leftrightarrow \exists n \geqslant i, (\pi,n) \vDash \psi \text{ et } \exists i_0, i_1, ..., i_k, \text{ avec } i_0 = i, i_k = n, \text{ et } i_{j+1} = succ^b(i_j) \\
&\quad \text{tels que } \forall j, 0 \leqslant j < k \Rightarrow (\pi, i_j) \vDash \varphi \\
(\pi,i) \vDash \neg(\varphi U^b \psi) &\Leftrightarrow \text{avec } (u_n)_{n\in\mathbb{N}} \text{ tel que } u_{n+1} = succ^b(u_n), u_0 = i, \\
&\quad (\forall n \in \mathbb{N}, (\pi, u_n) \vDash \neg\psi) \text{ ou } (\exists n', (\pi, u_{n'}) \vDash \neg\varphi \text{ et } \forall m < n', (\pi, u_m) \vDash \neg\psi) \\
(\pi,i) \vDash \neg\neg\varphi &\Leftrightarrow (\pi,i) \vDash \varphi
\end{aligned}$$

Les opérateurs généraux (X^g et U^g) permettent d'étudier l'intégralité du mot. Ils représentent les opérateurs classiques de LTL X et U. Les opérateurs abstraits (X^a et U^a) permettent d'étudier le

contenu du module dans lequel la modalité est considérée et les opérateurs du passé (X^- et U^-), permettent de représenter les propriétés sur les derniers appels effectués en remontant dans la pile d'appel des modules. De la même façon que pour LTL, nous définissons G^b et F^b avec $b \in \{a, g, -\}$.

Exemples de spécifications Retournons dans le cadre de la programmation concurrente. Posons x une variable modélisant le verrou sur une ressource ($x == 0$ si la ressource est libre, 1 sinon). Nous pouvons modéliser les propriétés de sûreté, contextuelles et de vivacité présentées dans l'introduction :

1. G^a $(x == 0)$, ou partout sur la trace abstraite partant du début de l'exécution, on a $x == 0$.
2. G^g $(\neg X^- x == 1)$, ou à tout moment, on ne pouvait pas avoir $x == 1$ lors du dernier empilement.
3. G^g $(F^a x == 0)$, ou à tout moment, $x == 0$ avant la terminaison du module courant.

4. Model-checking avec *CaRet*

Définition 4. *Soit P un ensemble de propositions, R une AMER sur P, et φ une formule CaRet. On dit que $R \vDash \varphi$ si et seulement si pour tout chemin π de R, $(\pi, 0) \vDash \varphi$.*

A partir d'une *AMER* R et d'une formule *CaRet* φ, est-ce que $R \vDash \varphi$? Afin de répondre à cette question centrale, nous présenterons un algorithme de vérification proche de celui décrit dans [2] auquel nous apportons plusieurs optimisations. Il sera composé de deux parties :

1. Construction d'un Automate de Büchi Généralisé Récursif (*ABGR*) S à partir d'une *AMER* R et la négation d'une formule *CaRet* φ
2. Test du vide sur S

4.1. Automate de Büchi Généralisé Récursif

Un Automate de Büchi Généralisé Récursif (*ABGR*) possède une structure similaire à une *AMER*, dans laquelle on remplace la fonction η par plusieurs conditions d'acceptation de Büchi. C'est un quadruplet $S = (M, \{S_m\}_{m \in M}, \mathcal{F}, init)$ avec $\mathcal{F} = \{F_1, F_2, ..., F_n\}$ une famille finie d'ensembles de nœuds de S. On dit qu'un chemin π est *acceptant* si pour chaque $F \in \mathcal{F}$, π passe infiniment souvent sur au moins un élément de F. Etant donné que F est fini, on peut dire qu'un chemin est acceptant si et seulement si pour chaque $F \in \mathcal{F}$, en notant $I(\pi)$ l'ensemble des éléments de π parcourus une infinité de fois, $I(\pi) \cap F \neq \emptyset$. On définit alors $\mathcal{L}(S)$ de la manière suivante :

$$\mathcal{L}(S) = \{\pi \in \Gamma^* \mid \pi \text{ est acceptant sur } S\}$$

4.2. Préliminaires

La première étape est de construire l'*ABGR* S à partir d'une *AMER* R et d'une formule *CaRet* φ. Par la suite, nous noterons $Atomes(\varphi)$ l'ensemble des atomes de φ, c'est à dire l'ensemble des ensembles de propositions atomiques vérifiables simultanément dans un état u, auxquelles on rajoute les formules qui en découlent [2]. Afin de simuler un parcours de l'automate, nous remplirons une pile avec deux informations : la boîte par laquelle l'exécution est passée et l'atome présent à l'entrée de la boîte. Une fois de retour dans le module initial, on pourra tester si l'atome de retour correspond bien à un successeur abstrait de l'atome empilé, ce qui nous permet de gérer la trace abstraite. De plus, il suffit de regarder l'atome empilé à partir de n'importe quel état pour connaître les propriétés vérifiées à l'appel du dernier module, donnant du sens aux modalités du passé.

La correspondance entre une *AMER* et l'*ABGR* associé se fonde sur la cohérence d'un atome A sur un état u, notée $u \Vdash A$. Dans le cas d'une *MER*, A est cohérent avec u si et seulement si $A \cap AP = \{p \mid \eta(u,p) = \top\}$. Si on a une approximation, quand $\eta(u,p) = \star$, les atomes peuvent inclure indifféremment p ou $\neg p$. On définit la cohérence telle que

$$u \Vdash A \Leftrightarrow \forall p \in AP, (\eta(u,p) = \top \Rightarrow p \in A) \wedge (\eta(u,p) = \bot \Rightarrow \neg p \in A)$$

Il nous faut également tester que, si un nœud possédant une formule de type $\varphi U^b \varphi'$ est rencontré, φ' finit par être vérifiée. C'est ici qu'intervient la condition de Büchi généralisée : on créera une condition de Büchi par formule de cette sorte. Toutefois, dans le cas $\varphi U^a \varphi'$, la condition de vivacité φ' doit être vérifiée dans la trace abstraite, dont le caractère infini est vital (si nous nous trouvons dans un module terminant à un état u, alors $succ^a(u) = \bot$). Il nous faut alors savoir si l'état courant se trouve dans une boucle infinie ou non. Enfin, si $b = -$, la trace est finie donc la vivacité n'est pas nécessaire.

4.3. Algorithme

4.3.1. Construction du *ABGR*

Soit $R = (M, \{R_m\}_{m \in M}, init)$ l'*AMER* et φ la formule à vérifier. L'*ABGR* que nous cherchons à construire est de la forme $S = (M', \{S_{m'}\}_{m' \in M'}, init', \mathcal{F})$. Soit $Tag = \{\texttt{inf}, \texttt{fin}\}$.

Modules On crée simplement un module pour chaque module existant dans l'*AMER* : $M' = M$. Pour chaque module $R_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m, \eta_m)$, nous construisons le module correspondant $S_m = (N'_m, B'_m, Y'_m, En'_m, Ex'_m, \delta'_m, \eta'_m)$.

Boîtes On munit chaque boîte b d'un atome A et d'un tag t. La nouvelle boîte (b, A, t) sera la boîte correspondant au module représenté par b, dont l'atome vrai à son entrée est A et un tag indiquant si l'exécution du module appelant b sera finie ou non. En empilant cette nouvelle boîte, on empile l'atome vérifié à l'appel du module.

$$\begin{cases} B'_m = B_m \times Atomes(\varphi) \times Tag \\ Y'_m(b, A, t) = Y_m(b) \end{cases}$$

Nœuds Dans l'algorithme de [2], l'ensemble des nœuds est directement $N'_m = \{(N_m \backslash Ex_m) \times Atomes(\varphi) \times Tag\} \cup \{Ex_m \times Atomes(\varphi) \times Atomes(\varphi)\}$ modulo le test de cohérence définie dans la section 4.2. Nous proposons de décaler certains tests afin d'identifier les états de l'*ABGR* menant vers un cul-de-sac pour éviter de les générer, ce qui réduit par la suite le nombre de tests à effectuer lors de la création des transitions. Nous obtenons ainsi un *ABGR* plus petit que s'il était généré naïvement. Outre la relation de cohérence, ces tests s'appuient sur les notions ci-dessous.

- L'atome A satisfait la *Condition de succession générale* de A' si pour chaque formule de la forme $X^g\varphi$ de A, $\varphi \in A'$. On notera cette condition $succGen(A, A')$. On définit de même la *Condition de succession abstraite* : $succAbs(A, A')$ si pour tout $X^a\varphi \in A$, $\varphi \in A'$.
- Les *formules d'appel* de A, notées $Appel(A)$ sont toutes les formules de la forme $X^-\varphi$ dans A.
- La fermeture de φ, ou $\Downarrow \varphi$, est l'ensemble des sous-formules de φ.

Les tests sont définis de la manière suivante.

- Si u n'est ni une sortie de module, ni un appel de module, ni un retour de module, alors on le transforme en (u, A, t), avec $A \in Atomes(\varphi)$ cohérent avec u et $t \in Tag$ signalant si le module dans lequel se trouve u termine ou non.
- Si u est une sortie ($u \in Ex_m$ pour un certain $m \in M$), on le transforme en (u, A, R), avec A et R deux atomes de φ tels que A est cohérent avec u et R est l'ensemble des propriétés vérifiées

au retour du module. Les deux atomes se "suivent" sur la trace générale ($succGen(A, R)$). De plus, u étant l'ultime état du module, il n'a pas de successeur abstrait. L'atome A ne doit donc pas contenir de formule de la forme $X^a\varphi$.

- Si u est un appel de module ($u = (b, e)$, avec b une boîte et e une entrée du module représenté par la boîte), on le transforme en $((b, A, t), (e, A', t'))$, où (b, A, t) est la nouvelle boîte appelée par u, A est cohérent avec u, et possède la propriété `call`. A' vérifie $succGen(A, A')$. De plus, il est clair que les formules d'appel de A' doivent être vérifiées dans A puisque c'est l'atome empilé. Enfin, si $t' = \texttt{inf}$, l'appel ne retournera jamais et il n'existera pas de successeur abstrait à (b, e). Là encore, dans ce cas, A ne doit pas contenir de formule $X^a\varphi$.
- Si u est un retour de module ($u = (b, x)$), on le transforme en $((b, A, t), (x, A', R))$. R est nécessairement cohérent avec u et possède la propriété `ret`. A étant l'atome vérifié à l'appel du module, on a également que $succAbs(A, R)$ et les formules d'appel de A et de R sont les mêmes (elles réfèrent à l'entrée du module qui a appelé b).

En tenant compte de ces considérations, il vient alors :

$$
\begin{aligned}
N'_m &= \{(u, A, t) | u \in N_m \backslash Ex_m; A \in Atomes(\varphi); \texttt{int} \in A; t \in Tag, A \Vdash u\} \cup Ex'_m \\
Ex'_m &= \left\{ (u, A, R) | \begin{array}{c} u \in Ex_m; A, R \in Atomes(\varphi); \texttt{int} \in A; u \Vdash A \\ \forall \Psi X^a \Psi \notin A; succGen(A, R) \end{array} \right\} \\
En'_m &= \{(u, A, t) | u \in En_m; A \in Atomes(\varphi); \texttt{int} \in A; u \Vdash A\} \\
Calls'_m &= \left\{ ((b, A, t), (e, A', t')) | \begin{array}{c} (b, e) \in Calls_m; A, A' \in Atomes(\varphi); \texttt{call} \in A, (b, e) \Vdash A \\ succGen(A, A'); Appel(A') = \{X^- \Psi \in \Downarrow \varphi; \Psi \in A\} \\ t' = \texttt{inf} \Rightarrow \forall \Psi; X^a \Psi \notin A \end{array} \right\} \\
Retns'_m &= \left\{ ((b, A, t), (x, A', R)) | \begin{array}{c} (b, x) \in Retns_m; A, A', R \in Atomes(\varphi); \texttt{ret} \in R, (b, x) \Vdash R \\ succAbs(A, R); Appel(A) = Appel(R) \end{array} \right\}
\end{aligned}
$$

Fonction de transition Les états et les retours ont une structure un peu complexe pour le traitement des transitions, aussi considérons qu'un retour $((b, A, t), (x, A', R))$ peut se lire comme $((b, x), R, t)$ et un appel $((b, A, t), (e, A', t'))$ comme $((b, e), A, t)$. Pour chaque transition existante (c'est à dire une relation du type $v \in \delta_m(u)$), pour que (v, A', T) soit inclus dans $\delta(u, A, t)$ (avec $T \in Atomes(\varphi)$ si v est une sortie et $T \in Tag$ sinon) on doit avoir :

- Les formules d'appel de A et de A' sont les mêmes.
- Les conditions de succession générales et abstraites de A sur A' sont vérifiées

Par ailleurs, considérons les *tags* portés par les nœuds. Ces derniers sont présents à titre indicatif afin de se souvenir si le module dans lequel se trouve le nœud va terminer ou non et ne requièrent pas de traitement préalable. Ils servent en réalité à vérifier que la trace d'exécution étudiée est infinie, et plus précisément à vérifier que les formules de type $\varphi U^a \varphi'$ sont vérifiées. Ce point sera précisé plus tard lorsque nous étudierons la création de la condition d'acceptation.

Dans le second automate, lorsque deux nœuds sont liés, leurs *tags* doivent être identiques. Implicitement, bien qu'un nœud de sortie (x, A, R) n'ait pas de *tag*, il se trouve dans un module qui termine puisqu'il en sort. Un nœud allant vers une sortie doit alors avoir un *tag* égal à `fin`. Enfin, si un nœud (u, A, t) se dirige vers un appel et que le module appelé ne termine pas $((b, A', t'), (e, A'', t''))$ avec $t'' = \texttt{inf}$), alors le module contenant (u, A, t) ne termine pas non plus et donc $t = \texttt{inf}$ et la trace abstraite partant de ce nœud prend fin. Formellement, les tests à effectuer sont :

1. **D'un nœud** (u, A, t) **vers un appel** $((b, A', t'), (e, A'', t''))$
   - $Appel(A) = Appel(A')$
   - $succGen(A, A')$
   - $t = t'$ et si $t'' = \texttt{inf}$, alors $t = \texttt{inf}$ et il n'y a pas de formule du type $X^a\varphi$ dans A'
2. **D'un nœud** (u, A, t) **vers une sortie** (x, A', R)
   - $Appel(A) = Appel(A')$

- $succGen(A, A')$
- $t = \texttt{fin}$

3. **D'un retour** $((b, A, t), (x, A', R))$ **vers un nœud** (u, A'', t'')
   - $Appel(R) = Appel(A'')$
   - $succGen(R, A'')$
   - $t'' = t$
4. **D'un nœud** (u, A, t) **vers un simple nœud** (u', A', t')
   - $Appel(A) = Appel(A')$
   - $succGen(A, A')$
   - $t' = t$

États initiaux et condition d'acceptation L'ensemble des nœuds initiaux est $init' = \{(u, A, t) \mid u \in init, \varphi \in A,\ Appel(A) = \emptyset,\ t = \texttt{inf}\}$. Enfin, un atome A *satisfait momentanément* une formule $\varphi_1 U^b \varphi_2$ ($b = g$ ou a) si dans cet atome, on a $\varphi_1 U^b \varphi_2 \Rightarrow \varphi_2$, autrement dit si $\varphi_2 \in A$ ou $\varphi_1 U^b \varphi_2 \notin A$. On définit la condition d'acceptation $\mathcal{F}$ contenant les ensembles suivants :

- Un ensemble contenant les états (u, A, t) où $t = \texttt{inf}$.
- Pour chaque formule $\varphi_1 U^b \varphi_2 \in Cl(\varphi)$, on crée l'ensemble des états (u, A, t), (x, A, R), $((b, A, t), (e, A', t'))$ ou $((b, A', t), (x, A'', A))$ dans lesquels A satisfait momentanément $\varphi_1 U^b \varphi_2$ avec $b = g$ ou a. Si $b = a$, on doit également avoir la condition $t = \texttt{inf}$.

Remarque : On n'a pas à regarder les sorties dans le cas abstrait, on sait déjà qu'elles n'ont pas de successeur abstrait.

4.3.2. Vérification de la condition

Afin de calculer $\mathcal{L}(S_{\neg\varphi})$, on parcourt l'*ABGR* généré et on cherche les boucles dans lesquelles l'ensemble des conditions d'acceptation est entièrement vérifié. On effectue un parcours complet de l'automate (parcours en profondeur) et on cherche l'ensemble des boucles vérifiant tout ou partie des conditions de Büchi, c'est à dire s'il passe par au moins un état acceptant de la condition. On fusionne alors les potentielles boucles croisées à l'intérieur d'une boucle, puis les boucles ayant le même point de départ. Cette méthode donne des résultats équivalente à celle de la fusion des conditions des conditions de Büchi [1], plus classique mais moins économique en espace.

4.4. Résultats

Cette construction nous permet de pouvoir analyser l'ensemble des chemins possibles de l'*AMER* R selon les propriétés associées à chaque nœud. Ainsi, il vient que :

Propriété 1. *Soit R une AMER et φ une formule CaRet. Si $\mathcal{L}(S_{\neg\varphi}) = \emptyset$, alors $R \vDash \varphi$.*

Démonstration. Si $R \nvDash \varphi$, il existe un chemin π ne satisfaisant pas φ. Lors de la génération de $S_{\neg\varphi}$, on suivra π en associant à chaque élément de π un atome contenant les propriétés vérifiées ou inconnues. La cohérence acceptera chaque élément de π, et le fera alors apparaitre au moins une fois dans $S_{\neg\varphi}$. □

En outre, il est possible de comparer deux *AMER* ayant les mêmes modules.

Définition 5. *Deux AMER $R = (M, \{R_m\}_{m\in M}, \eta, init)$ et $R' = (M', \{R'_m\}_{m\in M'}, \eta', init')$ sont dits* équivalentes *si $M = M'$, $\forall m \in M, R_m = R'_m$ et $init = init'$.*

Définition 6. *Soient R_1 et R_2 deux AMER. Notons $Ant_a(R) = \{(s, p) | \eta(s, p) = a\}$ les antécédents de a par η, avec $a \in \{\top, \bot, \star\}$ On dit que R_1* approxime *R_2, ou $R_1 \geqslant R_2$ si et seulement si :*

- R_1 *et* R_2 *sont équivalentes*
- $Ant_a(R_1) \subseteq Ant_a(R_2)$ *pour* $a \in \{\top, \bot\}$

Remarque On a par ailleurs $Ant_\star(R_2) \subseteq Ant_\star(R_1)$. En particulier, si R_2 est une *MER* classique, on a $Ant_\star(R_2) = \emptyset$ par définition, et une approximation R_1 de R_2 est toute *AMER* équivalente dont les propriétés non-$\star$ coïncident avec celles de R_2.

Propriété 2. *Soient AP un ensemble de propositions,* R_1 *et* R_2 *deux AMER sur AP telles que* $R_1 \geqslant R_2$*,* φ *une formule CaRet. Soient* S_1 *et* S_2 *les ABGR correspondant respectivement à* R_1 *et* R_2*. On a alors* $\mathcal{L}(S_2) \subseteq \mathcal{L}(S_1)$*.*

Démonstration. Si $R_1 \geqslant R_2$, alors $Ant_a(R_1) \subseteq Ant_a(R_2)$ pour $a \in \{\top, \bot\}$. Notons $N(S)$ l'ensemble des noeuds d'un *ABGR* S. Lors de la construction des *ABGR*, la fonction de cohérence va accepter et refuser, pour S_1 et S_2, les mêmes états lorsque les atomes associés n'ont pas de proposition *indécidées*. Or, comme $Ant_\star(R_2) \subseteq Ant_\star(R_1)$, plus d'états seront cohérents sur S_1 que sur S_2. La fonction de cohérence est la seule dont le résultat change entre l'ensemble des tests de la génération des *ABGR* S_1 et S_2 lorsque $R_1 \geqslant R_2$, car seule η change. Ainsi il vient que $N(S_2) \subseteq N(S_1)$, et donc $\mathcal{L}(S_2) \subseteq \mathcal{L}(S_1)$. □

Théorème 1. *Soient R une MER et* φ *une formule CaRet. S'il existe une AMER* R' *telle que* $R' \geqslant R$ *et* $\mathcal{L}(S'_{\neg\varphi}) = \emptyset$*, avec* $S'_{\neg\varphi}$ *l'ABGR associé à* R'*, alors* $R \vDash \varphi$

Démonstration. Immédiat d'après les propriétés 1 et 2. □

5. Application dans Frama-C

L'originalité de notre implantation est, outre de pouvoir vérifier une spécification de forme *CaRet* sur un programme codé en C, de proposer une interprétation d'un programme C en une *AMER* dont la précision dépend d'un oracle indépendant.

5.1. Interpréter le C en *AMER*

C Intermediate Language (CIL) [18] est un langage intermédiaire contenant une sous-partie du langage C. CIL a la même expressivité que C, mais propose des constructions plus simples à analyser. Ainsi, il n'y a qu'un seul type de boucle, et les expressions sont pures, seules les instructions ont des effets de bords. Un programme traité par Frama-C est automatiquement traduit en graphe de flot de contrôle (GFC) CIL. Posons la syntaxe des instructions du langage While, proche de CIL :

$i ::=$	$i; i$	(séquence)
	$\mid x :=$ "*expression*"	(affectation)
	$\mid$ if "*expression*" then i else i fi	(condition)
	$\mid$ while "*expression*" do i done	(boucle)
	$\mid f$("*expression*", ...)	(appel de fonction)
	$\mid$ return "*expression*"	(fin de fonction)

On peut définir une fonction comme une suite finie d'instructions. Une fonction C possède exactement une entrée, et CIL garantit qu'elle a également un unique point de retour.

Construction de la structure d'une *AMER* Soit AP un ensemble de propositions et P un programme composé de la famille de fonctions While $F = \{f_1, ...f_n\}$, avec f_1 la première fonction exécutée. On construit l'*AMER* $R = (M, \{R_m\}_{m \in M}, \eta, init)$ avec $R_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m)$ telle que : $M = \{1, ..., n\}$ et pour chaque $m \in M$ on parcourt chaque instruction de f_m :

```
int x=0 ;

void f(void) { x ++ ; x -- ; }

void g(void) { x = 1 ; x = 0 ; }

int main() { while(1){ if(!x) {f();} g(); }  return 0;}
```

FIGURE 4 – Exemple de programme

1. Séquence : On traite l'instruction gauche puis l'instruction droite.
2. Affectation ou retour de fonction : On ne fait rien de plus que créer le nœud et le relier au noeud précédent.
3. Condition : On traite chacune des instructions filles puis on relie chaque fille à la condition.
4. Boucle : On crée un nœud pour la boucle, on traite l'instruction de la boucle dont le dernier nœud est relié à la tête de la boucle
5. Appel de la fonction f_i : On crée une boîte b telle que $Y_m(b) = i$. Soit e l'entrée de f_i et x sa sortie. On crée les nœuds (b, e) et (b, x) entrées et sorties de b et on relie le nord d'entrée au noeud précédent.

Enfin, l'ensemble des états initiaux *init* est $\{n_1\}$, où n_1 est le nœud correspondant à la première instruction de f_1. Bien qu'il nous manque encore η, travail que nous déléguerons à un oracle, toutes les *AMER* construites telles que décrit précédemment seront équivalentes. Par exemple, à partir du programme en Figure 4, nous créerons l'automate en Figure 5.

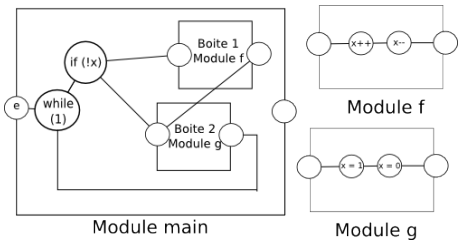


FIGURE 5 – *AMER* simulant le programme

Définition 7. *Soient AP un ensemble de propositions,* Γ *un alphabet, P un programme, R une AMER sur AP et* $p \in AP$*. Posons la notion de* configuration *comme étant un état mémoire du programme. Soit* $E = (c, ins)$ *un couple (configuration,instruction) de P, on dira que* $E \vDash p$ *lorsque la configuration associée satisfait p. On dit que P est simulé par R s'il existe une bijection entre* $Calls_m$ *et* $Retns_m$ *et si pour toute trace d'exécution* $\pi_P = \pi_P^1\pi_P^2...$ *finie ou non de P (avec* $\pi_P^i = (c_P^i, ins_P^i)$*), il existe un chemin* $\pi_R = \pi_R^1\pi_R^2...$ *de R sur* $N \cup Calls$ *tel que pour tout* $i < |\pi_P|$ *:*

Pour tout $E \in \{\pi_P^j | \pi_R^j = \pi_R^i\}$*, si* $\eta(\pi_R^i, p) = \top$*, alors* $E \vDash p$ *et si* $\eta(\pi_R^i, p) = \bot$*, alors* $E \nvDash p$

Lorsque, pour une propriété p et deux configurations $c_P^\top$ et $c_P^\bot$ possibles sur une même instruction, c'est à dire $c_P^\top \vDash p$ et $c_P^\bot \vDash \neg p$, η renverra $\star$.

Définition 8. *Soit P un programme While,* φ *une formule CaRet. On dit que* $P \vDash \varphi$ *si pour toute trace d'exécution* π *de P,* $\pi \vDash \varphi$*.*

Propriété 3. *S'il existe une AMER R simulant P tel que* $R \vDash \varphi$*, alors* $P \vDash \varphi$

Démonstration. Si $P \not\models \varphi$, alors on peut trouver une trace d'exécution π_P ne satisfaisant pas φ. On a donc d'après la définition 7 une trace π_R ne satisfaisant pas la formule et donc $R \not\models \varphi$. □

Théorème 2. *Soit AP un ensemble de propositions. Soit un oracle $\eta : AP \to instr \to \{\top, \bot, \star\}$ évaluant si une proposition p de AP est vérifiée en une instruction i. Soit $R = (M, \{R_m\}_{m \in M}, \eta, init)$ une AMER simulant P. Si $\mathcal{L}(S_{\neg\varphi}) = \emptyset$, avec $S_{\neg\varphi}$ l'ABGR associé à R et $\neg\varphi$, alors $P \models \varphi$.*

Un bon candidat pour l'oracle est le greffon Value [8, 10], outil d'interprétation abstraite calculant l'ensemble des états possibles pris par le programme en chaque instruction. La prémisse de notre implantation consiste à parcourir le GFC du programme CIL avec Value et de sauvegarder les résultats calculés. Une fois que le programme est sous forme d'un GFC traité par Value, on peut travailler dessus comme si on travaillait sur une *AMER*.

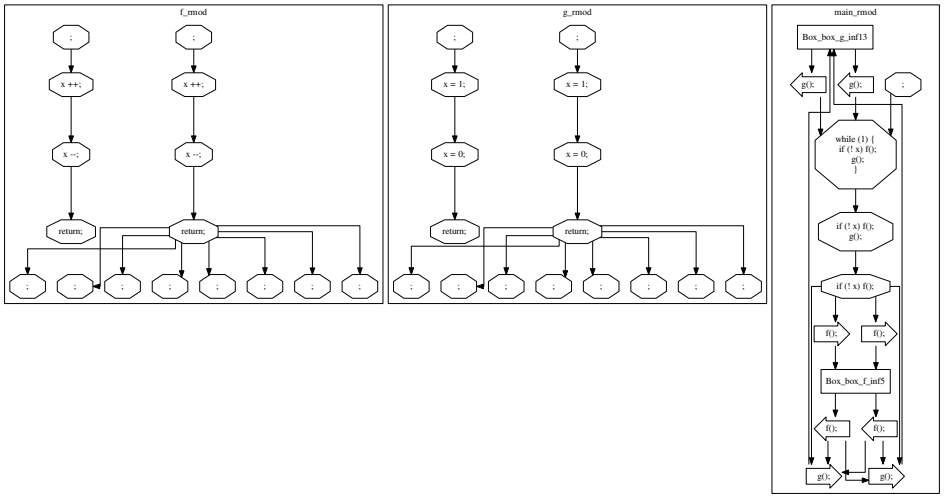
Notons par ailleurs une abstraction supplémentaire introduite par la construction décrite précédemment car toutes les boucles dans un programme ne sont pas infinies, alors qu'une *AMER* ne fait pas de distinction entre les boucles. Nous perdons donc de la complétude sur le résultat puisqu'il peut être possible de trouver un cycle acceptant dans l'*AMER* alors qu'il s'agit d'une boucle finie pour le programme initial et dont possiblemment renvoyer un faux positif.

5.2. Application de *CaFE*

Reprenons une nouvelle fois le programme en Figure 4. Chaque fonction du programme (sauf le main) modifie x. On souhaite vérifier l'accessibilité de la ressource de manière plus ou moins précise :

1. $G^a(x == 0)$: À tout moment dans le module principal, la ressource doit être accessible.
2. $G^g(x == 0)$: À tout moment, la ressource doit être accessible.

Dans les deux cas, $AP = \{x == 0\}$. Nous générons à présent les deux *ABGR* S_1 et S_2 modélisant les possibles exécutions du programme ne satisfaisant pas la formule, autrement dit satisfaisant leur contraire. Ils auront une structure très similaire ; seul le nombre d'états acceptants change.



Le module main, à droite, appelle successivement le module f et g avant de revenir au début de la boucle while. Les deux paires de branches à gauche représentent respectivement les modules f et g.

FIGURE 6 – *ABGR* créé via l'*AMER* précédent et la formule 1

On normalise la formule de manière à n'avoir que les opérateurs X et U. Rappelons que $G^b\varphi = \neg(\top\ U^b\ \neg\varphi)$. Les *ABGR* auront alors pour condition d'acceptation :

S_1 : $\mathcal{F} = \{\{(u, A, t)|t = \texttt{inf} \wedge (\neg\varphi \in A \text{ ou } \top\ U^a\ \neg(x == 0) \notin A)\}\}$

S_2 : $\mathcal{F} = \{\{(u, A, t)|t = \texttt{inf}\}; \{(u, A, t)|\neg\varphi \in A \text{ ou } \top\ U^g\ \neg(x == 0) \notin A\}\}$

Pour le premier *ABGR*, $\top\ U^a\ \neg(x == 0)$ est vérifié aux mêmes endroits, dans les fonctions f et g, mais les tags sont alors `fin` puisque l'exécution de ces deux modules sont finis. Ainsi, la condition de Büchi étant l'ensemble vide elle ne peut pas être vérifiée, à l'instar de la négation de la formule de base. La formule est alors vérifiée.

```
Your program satisfies the formula !
```

Pour le second, dès le début de la boucle on entre dans un état taggé `inf` puisque l'exécution du module principal est infini. Or, à l'intérieur de f, on repère la propriété $\neg(x == 0)$, satisfaisant momentanément la formule $\top\ U^g\ \neg(x == 0)$. Le langage accepté par l'automate n'est pas vide, donc la formule est fausse et un contre-exemple est généré. Notons cependant que dans le cas général, ce contre-exemple pourrait n'être qu'une fausse alarme liée aux approximations effectuées.

```
A path has been found that doesn't satisfy the formula !

-->";_st_26Inf: entry3_inf" // Etat d'entree
--> LOOP //Debut de la boucle
-->"while (1) { if (! x) f(); g();}
_st_61Inf: int_99_inf"
--> ... // Etats dans la boucle
--> LOOP //De retour au debut de la boucle
-->"while (1) {if (! x) f(); g();}
_st_61Inf: int_99_inf"
```

6. Conclusion et perspectives

Nous avons présenté ici une extension de la logique *CaRet* qui facilite son utilisation au sein de Frama-C en collaboration avec les greffons d'analyse existants, et en particulier Value, afin d'introduire à la logique temporelle un phénomène d'incertitude. Cette logique permet d'exprimer, et de vérifier, des propriétés portant sur l'ensemble d'une trace d'exécution d'un programme C infinies, ce qui est difficile à réaliser en ACSL pur. Par ailleurs, diverses optimisations ont été apportées pour limiter le nombre d'états à considérer lors de l'analyse. Ces différents points ont en outre été implantés au sein de Frama-C dans le greffon *CaFE*. Des perspectives d'améliorations de *CaFE* subsistent cependant.

La précision de l'oracle est un point crucial de notre technique. Plusieurs développements sont possibles à ce sujet. Tout d'abord, il est possible de faire correspondre plusieurs nœuds de l'*AMER* à une même instruction du programme, ce qui revient par exemple à dupliquer des fonctions ou à dérouler des boucles. Cela permettrait à la fonction η d'avoir plus de sensibilité au contexte d'exécution, et de tirer parti de certaines options de Value qui vont dans le même sens. En pratique, cela rendra d'autant plus important le contrôle de l'explosion combinatoire, puisque l'automate d'origine comportera un plus grand nombre de nœuds. Il serait également possible d'utiliser d'autres greffons, comme par exemple WP [10], a priori plus puissant pour étudier des propriétés arbitraires, mais d'un maniement plus délicat, pour établir les propriétés qui nous intéressent. Enfin, il serait intéressant de ne pas se reposer entièrement sur l'oracle, mais de pouvoir propager des informations sur l'état du programme directement dans l'*AMER*. Ce dernier point permettrait de faire collaborer étroitement interprétation abstraite (avec Value) et model-checking pour la preuve de propriétés temporelles, mais nécessite une représentation de la mémoire et des pointeurs C dans nos *AMER*.

Un second axe de recherche concerne l'adaptation de *CaRet* à certains traits C/C++. Tout d'abord, il s'agirait de traiter les pointeurs de fonction. Bien qu'indéfinis dans les *AMER*, on peut imaginer une

boîte dont le module auquel elles se réfèrent est une variable, mais la définition des chemins possibles dans une telle *AMER* devient délicate. De même, le traitement des exceptions, ou de son équivalent C sous forme de `setjmp/longjmp`, aurait un impact sur la représentation de la pile d'appel, puisqu'un événement `call` pourrait dès lors ne pas être suivi d'un événement `ret` correspondant.

Références

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM TOPLAS*, 27(4), 2005.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481. Springer, 2004.

[3] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7) :68–76, July 2011.

[4] T. Ball and S. K. Rajamani. *Slic : a Specification Language for Interface Checking (of C)*. Microsoft Research, Jan. 2002.

[5] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2014. http://frama-c.com/download/acsl.pdf.

[6] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming*, pages 652–666. Springer, 2001.

[7] D. Beyer and M. E. Keremoglu. CPAchecker : A Tool for Configurable Software Verification. In *Proceedings of CAV (Computer Aided Verification)*, volume 6806 of *LNCS*, pages 184–190, 2011.

[8] G. Canet, P. Cuoq, and B. Monate. A value analysis for c programs. In *Source Code Analysis and Manipulation. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 2009.

[9] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422. Springer, 1996.

[10] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. In *International Conference on Software Engineering and Formal Methods (SEFM'12)*. Springer, Oct. 2012.

[11] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247. Springer, 2000.

[12] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification*, pages 53–65. Springer, 2001.

[13] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java modeling language. *IET Software*, 2(6) :500–514, Dec. 2008.

[14] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 1997.

[15] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41, Oct. 2009.

[16] F. Merz, S. Falke, and C. Sinz. LLBMC : Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *VSTTE*, volume 7152 of *LNCS*, 2012.

[17] B. Meyer. Applying "design by contract". *Computer (IEEE)*, 25(10), Oct. 1992.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil : Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*. Springer, 2002.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[20] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science (1977)*, pages 46–77, 1977.

[21] A. N. Prior. Time and modality. *Clarendon Press*, 1957.

[22] N. Stouls and V. Prevosto. *Aoraï plug-in tutorial, version Nitrogen-20111001*, Oct. 2011. http://frama-c.com/download/frama-c-aorai-manual.pdf.