Stratégies Dynamiques pour la Génération de Contre-exemples

Nguyen Le Vinh, Hélène Collavizza, Michel Rueher, Samuel Devulder, Thierry Gueguen

Stratégies Dynamiques pour la Génération de Contre-exemples*

Le Vinh Nguyen[1] **Hélène Collavizza**[1] **Michel Rueher**[1]
Samuel Devulder[2] **Thierry Gueguen**[2]

[1] Université de Nice-Sophia Antipolis/CNRS, 2000, route des Lucioles - Les Algorithmes
BP 121 - 06903 Sophia Antipolis Cedex - France
[2] Geensys, 120 Rue René Descartes, 29280 Plouzané - France
{lvnguyen, helen, rueher}@polytech.unice.fr
{samuel.devulder, thierry.gueguen}@geensys.com

Résumé

La vérification de propriétés de sûreté est incontournable dans le processus de validation des logiciels critiques. Lorsque les outils de vérification formelle échouent à démontrer certaines propriétés, la recherche de contre-exemples des propriétés violées est un point crucial, notamment pour les programmes complexes pour lesquels les jeux de test sont difficiles à générer. Nous proposons dans cet article différentes stratégies dynamiques de génération de jeux de test qui violent une post-condition de programmes écrits en C ou en Java. Notre approche est basée sur la génération dynamique d'un système de contraintes lors de l'exploration du graphe de flot de contrôle du programme. Ces stratégies ont été évaluées sur des exemples académiques et sur des applications réelles. Les expérimentations sur un contrôleur industriel qui gère les feux de clignotants d'une voiture et sur une implémentation d'un système de détection de collisions du trafic aérien ont montré que notre système est bien plus performant qu'un outil de model checking de premier plan comme CBMC ou qu'un système de génération de jeux de test à base de contraintes comme Euclide.

Abstract

Checking safety properties is mandatory in the validation process of critical software. When formal verification tools fail to prove some properties, testing is necessary. Generation of counterexamples violating some properties is therefore an important issue, especially for tricky programs the test cases of which are very difficult to compute. We propose in this paper different constraint based dynamic strategies for generating structural test cases that violate a post-condition of C or JAVA programs. These strategies have been evaluated on standard benchmarks and on real applications. Experiments on a real industrial Flasher Manager controller and on the public available implementation of the Traffic Collision Avoidance System (TCAS) show that our system outperforms state of the art model checking tools like CBMC or constraint based test generation systems like Euclide.

1 Introduction

La vérification de programmes fait appel à des techniques de preuves formelles, des techniques de tests fonctionnels et structurels, des revues et analyses manuelles de code. Lorsque les outils de vérification formelle échouent à prouver les post-conditions d'un programme, ce travail est souvent fait à la main. De plus, la vérification formelle se base sur un modèle abstrait qui ne capture pas toujours toutes les fonctionnalités de l'implémentation, et qui est souvent de taille limitée. Aussi, le test est toujours nécessaire, et la génération automatique de contre-exemples d'une propriété reste un point crucial dans le processus de vérification des programmes. En particulier, pour les applications temps réel, la génération de cas de test pour des périodes de temps réalistes reste un problème difficile.

Dans cet article, nous proposons de nouvelles stratégies dynamiques pour générer des cas de test structurel qui violent une post-condition de programmes écrits en C ou en Java[1]. Notre approche repose sur les observa-

*Ce travail a été partiellement soutenu par le projet CAVERN (ANR-07-SESUR-003), ainsi que par le projet TESTEC (ANR-07 TLOG 022).

[1] Nous traitons actuellement un sous-ensemble de ces lan-

tions suivantes :

- lorsque le programme est sous une forme DSA[2], un chemin d'exécution peut être construit de manière dynamique. En d'autres termes, il n'est pas nécessaire d'explorer le GFC (Graphe de Flot de Contrôle) du programme de manière séquentielle (descendante ou ascendante), mais les blocs compatibles sur un chemin peuvent être collectés de manière non déterministe.
- une partie significative du programme peut n'avoir aucun impact sur une propriété donnée[3]. En particulier, les parties du programme qui ne contiennent pas de variables de la post-condition, ni de variables liées aux variables de la post-condition, peuvent être ignorées quand on cherche un contre-exemple.

Dans l'approche préconisée, nous utilisons d'abord une technique classique, connue sous le nom de *slicing*, pour supprimer de façon statique les blocs du programme où n'apparaissent pas les variables pertinentes pour la propriété considérée. Nous explorons ensuite ce GFC réduit en suivant une variable particulière de la post-condition. Toutefois, il ne s'agit pas d'un parcours arrière chronologique : nous construisons les chemins de manière dynamique pour détecter au plus tôt des chemins inconsistants, c'est à dire sans examiner l'ensemble des noeuds desdits chemins. Par contre, en cas d'erreur, tous les noeuds correspondant au chemin du contre exemple doivent être examinés.

Concrètement la stratégie *DPVS* (Dynamic Postcondition-Variables driven Strategies) travaille avec un ensemble de contraintes S et une file de variables. Nous avons essayé différentes gestions de la file des variables : FIFO, LIFO et TAS (trié par rapport au niveau des variables dans le GFC). Les meilleurs résultats ont été obtenus avec les stratégies LIFO et TAS. Nous avons aussi essayé différentes combinaisons de solveurs de contraintes sur les domaines finis (CP) et de solveurs de programmation linéaire (LP). Aucune de ces combinaisons n'est systématiquement meilleure qu'une autre mais les résultats sont assez prévisibles : lorsque les domaines des variables sont petits, CP se comporte mieux alors que LP est généralement meilleur si le programme a beaucoup d'expressions linéaires. Nous avons évalué *DPVS* sur des benchmarks standards ainsi que sur deux applications réelles :

- l'implémentation d'un système de gestion des collisions de trafic aérien (TCAS) ;
- une application réelle industrielle, un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule, appelée *Clignotant* dans la suite.

Sur ces applications réelles, *DPVS* surpasse des outils de model checking de premier plan comme CBMC ainsi qu'un système de génération de test basé sur les contraintes comme Euclide.

1.1 Etat de l'art

Les outils récents de génération de tests structurels orientés chemins (par exemple, PathCrawler[20], Dart[13], CUTE[4]) sont basés sur la sélection de chemin, l'exécution symbolique et l'exécution concolitique. Ils sont très efficaces pour générer des jeux de test qui garantissent une certaine couverture, mais ils n'ont pas été conçus pour trouver les données de test qui violent certaines propriétés ; ils pourraient néanmoins le faire en effectuant une recherche exhaustive, mais ceci serait très coûteux.

Les model-checkers bornés [11], comme CBMC[5], peuvent trouver des contre-exemples de propriétés de programmes C[12, 17]. Ces outils transforment le programme et la post-condition en une formule conditionnelle et utilisent les solveurs SAT ou SMT pour prouver que cette formule est satisfaite ou pour trouver un contre-exemple.

La Programmation Logique par Contraintes (CLP) permet d'implémenter des techniques d'exécution symbolique [4], et a été utilisée pour la génération de tests de programmes (e.g., [16, 18, 21, 1]). Gotlieb et al ont montré comment transformer des programmes impératifs en programmes CLP : InKa [16] a été un pionnier dans l'utilisation de CLP pour générer des données de test pour les programmes en C. Denmat et al ont développé TAUPO, un successeur de InKa qui utilise des relaxations linéaires dynamiques [10]. Cela accroît la capacité de résolution du solveur en présence de contraintes non-linéaires, mais les auteurs ne publient des résultats expérimentaux que sur quelques programmes académiques en C.

Euclide [14] est également un successeur de InKa. Il a trois fonctions principales : la génération de données de test structurel, la génération de contre-exemples et la preuve partielle des programmes C critiques. Euclide construit les contraintes de manière incrémentale et combine des techniques standards de programmation par contraintes et des techniques spécifiques pour

gages incluant les entiers et les tableaux.

[2]La forme DSA (Dynamic Single Assignment)[2] est une représentation intermédiaire inspirée de la forme SSA bien connue en compilation. La forme DSA est une transformation qui préserve la sémantique du programme tout en assurant que chaque variable sur un chemin d'exécution ne soit définie qu'une seule fois.

[3]Nous supposons ici que la post-condition est une conjonction de propriétés.

[4]cf. `http://osl.cs.uiuc.edu/~ksen/cute/`

[5]cf. `http://www.cprover.org/cbmc`

traiter les nombres à virgule flottante et les contraintes linéaires.

CPBPV [6, 7, 8] est un outil basé sur les contraintes dont le but est de vérifier la conformité d'un programme avec ses spécifications. L'idée clef de CPBPV est d'utiliser un ensemble de contraintes pour représenter le programme et ses spécifications, et d'explorer de manière non-déterministe les chemins d'exécution de longueur bornée sur cet ensemble de contraintes. CPBPV fournit un contre-exemple lorsque le programme n'est pas conforme à sa spécification.

Les stratégies de recherche d'Euclide et de CPBPV ne sont pas bien adaptées à la recherche d'un contre-exemple. En effet, CPBPV est fondé sur une exploration descendante des chemins car il a été conçu pour la vérification partielle des programmes. Euclide - qui a été conçu pour la génération de données de test - explore dynamiquement des alternatives faisables mais utilise aussi une exploration descendante. Ces stratégies peuvent donc devenir très coûteuses lorsque le but est seulement de trouver un contre-exemple sur un programme complexe. Par contre, *DPVS* est une stratégie ascendante dynamique qui a été conçue pour trouver des contre-exemples.

1.2 Plan du papier

La section 2 illustre notre approche sur un petit exemple et introduit les nouveaux algorithmes de recherche que nous avons définis. La section 3 décrit les benchmarks et les applications que nous avons utilisées pour valider notre approche. La section 4 présente les résultats expérimentaux et les perspectives.

2 DPVS, la nouvelle stratégie de recherche à base de contraintes

Dans cette section, nous décrivons en termes très généraux les principes de notre approche et le processus de recherche sur un exemple simple. Ensuite, nous détaillons l'algorithme.

2.1 Un petit exemple

La stratégie *DPVS* construit incrémentalement un système de contraintes S et gère une file de variables Q. Q est initialisée avec les variables de la propriété de la post-condition pour laquelle nous cherchons un contre-exemple, alors que S est initialisé avec la négation de cette propriété. Tant que Q n'est pas vide, *DPVS* extrait la première variable v et cherche un bloc du programme où la variable v est définie. Toutes les nouvelles variables (sauf les variables d'entrée du programme) de cette définition sont mises dans Q. La définition de la variable v, ainsi que les conditions nécessaires pour parvenir à la définition de v sont ajoutées à S. Si S est inconsistant, *DPVS* fait un retour arrière et cherche une autre définition de v, sinon les conditions duales aux conditions ajoutées à S sont coupées pour éviter de perdre du temps à explorer des chemins incompatibles. Lorsque Q est vide, le solveur de contraintes est utilisé pour chercher un ensemble de valeurs des variables d'entrée qui violent la propriété, c'est-à-dire un contre-exemple.

Nous illustrons maintenant ce processus sur un exemple très simple, le programme foo présenté dans la figure 1. Ce programme comporte deux post-conditions : $p_1 : c >= d + e$ et $p_2 : f > -e * b$.

Le GFC du programme foo est présenté dans la figure 2. Notons que le programme a été transformé en une forme DSA[6]. Supposons que nous voulons prouver la propriété p_1. Avant de chercher un contre-exemple, nous calculons la forme compactée du GFC, où les nœuds qui ne sont pas liés à la propriété à prouver sont supprimés (cf. figure 3).

Les figures 4 et 5 présentent les chemins explorés par *DPVS*. Le processus de recherche sélectionne d'abord le nœud (4) où la variable c_0 est définie. Pour atteindre le nœud (4), l'état dans le nœud (0) doit être vrai. Cette condition est donc ajoutée à l'ensemble de contraintes S et le branchement alternatif est coupé. A cette étape, S contient les contraintes suivantes : $(c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = a_0 \wedge a_0 \geq 0)$ qui peut être simplifié en $(a_0 < 0 \wedge a_0 \geq 0)$. Cet ensemble de contraintes est inconsistant et *DPVS* sélectionne donc le nœud (8) où la variable c_0 est également définie. Pour atteindre le nœud (8), la condition dans le nœud (0) doit être fausse. La négation de cette condition est donc ajoutée à S et l'autre branche est coupée. S contient donc les contraintes suivantes : $(c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0)$ qui peut être simplifié à $(a_0 < 0 \wedge b_0 < 0)$. Cet ensemble de contraintes est consistant et le solveur calcule une solution, par exemple, $(a_0 = -1, b_0 = -1)$. Ces valeurs des variables d'entrée sont un jeu de test qui viole la propriété p_1 du programme foo.

Ce petit exemple montre comment fonctionne *DPVS* : cette nouvelle stratégie recueille le maximum d'informations sur les variables qui influencent la post-condition afin de détecter les inconsistances le plus tôt possible ; ceci est particulièrement efficace quand un petit sous-ensemble du système de contraintes est inconsistant.

[6]Nous ne considérons ici que des programmes bornés où la taille des tableaux, et le nombre d'itérations des boucles sont limités, nous simplifions des $\phi-$fonctions lors de construction de la forme SSA.

```
void foo(int a, int b)
1.  int c, d, e, f;
2.  if(a >= 0) {
4.      if(a < 10) {
5.          f = b − 1;
6.      }
7.      else {
8.          f = b − a;
9.      }
10.     c = a;
11.     if(b >= 0) {
12.         d = a; e = b;
13.     }
14.     else {
15.         d = a; e = −b;
16.     }
17. }
18. else {
19.     c = b; d = 1; e = −a;
20.     if(a > b) {
21.         f = b + e + a;
22.     }
23.     else {
24.         f = e ∗ a − b;
25.     }
26. }
27. c = c + d + e;
28. assert(c >= d + e); // property p1
29. assert(f >= −b ∗ e); // property p2
```

FIG. 1 – Programme *foo*

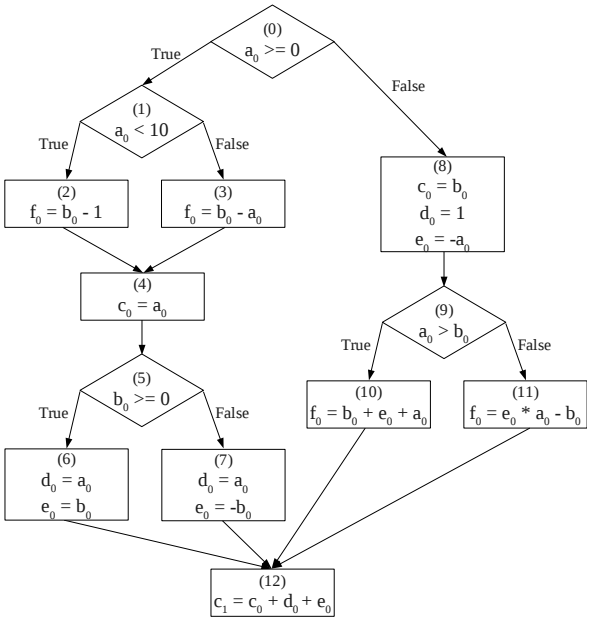

FIG. 2 – GFC du programme *foo* en forme DSA

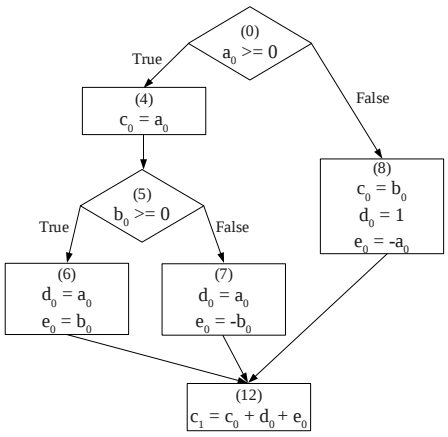

FIG. 3 – GFC compacté pour la propriété p_1

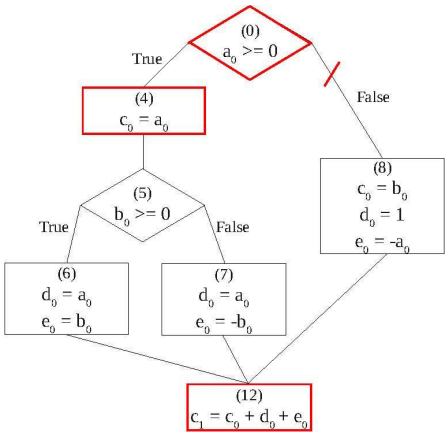

FIG. 4 – Processus de recherche pour p_1, étape 1

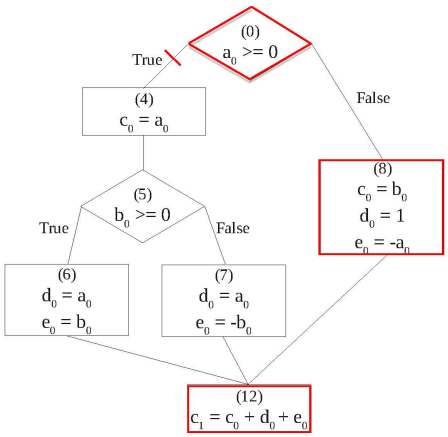

FIG. 5 – Processus de recherche pour p_1, étape 2

2.2 Algorithme

Nous détaillons ici l'algorithme *DPVS* (cf. figure 2.2). Nous calculons au préalable les informations suivantes :

- $du[x]$: l'ensemble des blocs où la variable x est définie ;
- $anc_c[u]$: l'ensemble des prédécesseurs de u qui sont des nœuds conditionnels ;
- $dr[u, v]$: un booléen qui est vrai (resp. faux) lorsque la condition du prédécesseur v du nœud u doit être vrai (resp. fausse) pour atteindre u.

DPVS utilise également les structures de données suivantes :

- M : l'ensemble des variables marquées (une variable est marquée si elle a déjà été mise dans la file) ; M est initialisé avec $\emptyset$;
- S : l'ensemble des contraintes qui est initialisé avec $const(pred \wedge \neg(prop))$ où *const* est une fonction qui transforme une expression sous forme DSA en un ensemble de contraintes ;
- Q : l'ensemble des variables temporaires, initialisé par $V(prop)$; l'ordre de traitement des éléments de Q est spécifié par le paramètre O_q

DPVS sélectionne une variable dans Q et essaie de trouver un contre-exemple avec sa première définition, si elle échoue, il essaie de manière itérative avec les autres définitions de la variable sélectionnée.

DPVS définit la couleur du nœud conditionnel u à rouge (resp. bleu) lorsque l'état de u est défini à true (false) dans le chemin courant. En d'autres termes, lorsque la couleur est rouge (resp. bleu) le branchement à droite (resp. gauche) de u est coupé. $color[u]$ est initialisé à vide pour tous les nœuds.

DPVS retourne *sol* qui est soit une instance des variables d'entrée de P satisfaisant le système de contraintes S ou $\emptyset$ si S n'a pas de solution. Les solutions sont calculées par la fonction *solve*, un solveur sur les domaines finis. La fonction *solve* est une procédure de décision complète sur les domaines finis. La fonction $isfeasible$ (ligne 27) effectue uniquement un test de la consistance partielle mais elle est beaucoup plus rapide que la fonction *solve*, ce qui justifie notre choix puisque ce test est effectué à chaque fois que la définition d'une variable est ajoutée à S.

Il est facile de montrer que *sol*, la solution calculée par *DPVS* est un contre-exemple. En effet, ces valeurs des données d'entrées satisfont les contraintes générées à partir de :

- *pred*, la pre-condition ;
- $\neg prop$, la négation d'une partie de la post-condition ;
- la définition de toutes les variables dans $V(prop)$ et la définition de toutes les variables (sauf les variables d'entrée), introduites par ces définitions ;
- toutes les conditions nécessaires pour atteindre les définitions ci-dessus.

Ainsi, il existe au moins un chemin exécutable qui prend des valeurs d'entrée *sol* et calcule une sortie qui viole la propriété *prop*. Lorsque aucune solution ne peut être trouvée, tous les chemins sont contradictoires avec la négation de la post-condition, et nous pouvons donc affirmer qu'il n'existe pas de valeurs d'entrée qui violent la propriété *prop*.

Algorithm 1 : *DPVS*

Function $DPVS(M, S, Q, O_q)$
returns *counterexample*

```
1:  if Q = ∅ then
2:    return solve(S)
3:  else
4:    x ← POP(Q, O_q)
5:    for all u ∈ du[x] do
6:      Cut ← FALSE ; SAVE vector Color
7:      S_1 ← S∧ const(def[x, u])
              % def[x, u] denotes the definition of x in
        block u
8:      V_new ← V(def[x, u]) \ M
9:      PUSH(Q, V_new, O_q) ; add(V_new, M)
10:     for all v ∈ anc_c[u] do
11:       if color[v] = blank then {%no branch is cut
          off}
12:         V_new ← V(condition[v]) \ M
13:         PUSH(Q, V_new, O_q) ; add(V_new, M)
14:         if dr[u, v] then {% Condition must be true}
15:           S_1 ← S_1∧ cons(condition[v]))
16:           color[v] ← red % Cut the right branch
17:         else {% Condition must be false}
18:           S_1 ← S_1 ∧ ¬ cons(condition[v])
19:           color[v] ← blue % Cut the left branch
20:         end if
21:       else
22:         if (color[v] = red ∧ dr[u, v])
                    ∨ (color[v] = blue ∧ ¬(dr[u, v]))
            then {%no branch is reachable}
23:           Cut ← TRUE
24:         end if
25:       end if
26:     end for
27:     if ¬Cut ∧ isfeasible(S_1) then
28:       result ← DPVS(M, S1, Q, O_q)
29:       if result ≠ ∅ then
30:         return result
31:       end if
32:     end if
33:     RESTORE vector Color
34:   end for
35:   return ∅
36: end if
```

3 Expérimentations et Applications

Dans cette section, nous décrivons les exemples et applications utilisés pour évaluer *DPVS*. Il s'agit d'exemples académiques, d'une implémentation du système de gestion des collisions de trafic aérien (TCAS), et enfin d'une application industrielle réelle, un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule.

3.1 Exemples académiques

3.1.1 Programme tritype

Le programme tritype est un benchmark standard pour la génération de cas de test et pour la vérification de programmes car il contient de nombreux chemins infaisables : seulement 10 chemins correspondent à des entrées réelles du programme, en raison de la complexité des instructions conditionnelles. Ce programme prend en entrée trois entiers positifs (les côtés d'un triangle) et retourne la valeur 1, (resp 2, 3) si les entrées correspondent à un triangle scalène (resp. isocèle, équilatéral) et la valeur 4 s'il ne s'agit pas d'un triangle.

Nous considérons aussi deux variations du programme *Tritype* :

1. *Triperimeter* qui retourne le périmètre du triangle lorsque les entrées correspondent à un triangle, sinon il retourne -1.
2. *Tritimes* qui retourne le produit des entrées.

Ces deux variations sont plus difficiles que le programme *Tritype* lui-même : *Triperimeter* retourne une expression linéaire sur les entrées et *Tritimes* retourne une expression **non** linéaire alors que*Tritype* retourne une constante.

Pour ces trois programmes, nous avons également examiné certaines versions où une erreur a été introduite.

3.1.2 Recherche binaire

Le deuxième exemple illustre le cas des programmes contenant des tableaux et des boucles. Il s'agit d'un programme de recherche binaire qui détermine si une valeur v est présente dans un tableau trié t.

3.2 Système anti-collisions de trafic aérien

Cette application concerne un composant logiciel connu du système anti-collisions de trafic aérien (TCAS). Il s'agit d'un système critique embarqué destiné à éviter les collisions en vol entre aéronefs [5]. Nous considérons ici le programme et la spécification qui ont été écrits par Arnaud Gotlieb [14, 15] à partir d'une version préliminaire donnée dans [19]. Nous résumons ici leurs caractéristiques principales.

Ce programme contient 173 millions de lignes de code C, y compris les conditions imbriquées, les opérateurs logiques, des définitions de type, les macros et les appels de fonction. La fonction principale à vérifier prend 14 variables globales comme entrées. 10 propriétés ont été identifiées et formalisées dans la littérature (cf. [14] pour une présentation complète des ces propriétés).

3.3 Clignotant

Ce dernier benchmark est un composant logiciel industriel temps-réel[7]. Il illustre la façon dont nous traitons les propriétés sur plusieurs unités de temps. La complexité du code C est comparable à la complexité du benchmark TCAS, mais nous devons vérifier une propriété pour plusieurs exécutions du code.

3.3.1 Description du module

Le *Clignotant* est un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule. La figure 6 fournit un modèle Simulink simplifié de ce contrôleur. Les entrées et sorties du modèle sont :

- CBSW_HAZARD_L : Activation du clignotant gauche
- CBSW_HAZARD_R : Activation du clignotant droit
- WARNING : Activation des warning
- RF_KEY_LOCK : Fermeture du véhicule par la clé
- RF_KEY_UNLOCK : Ouverture du véhicule par la clé
- FLASHER_ACTIVE : Fonction clignotant activée
- CMD_FLASHER_L : Commande du clignotant gauche
- CMD_FLASHER_R : Commande du clignotant droit

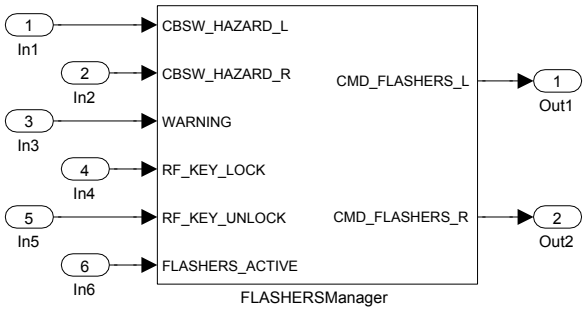

FIG. 6 – Modèle Simulink simplifié du *Clignotant*

[7] Cet exemple provient d'un constructeur automobile et a été fourni par Geensys (cf. `http://www.geensys.com/?Home&l=en`)

Certaines fonctions du *Clignotant* sont prioritaires sur d'autres. Nous avons cherché à vérifier la propriété p_1 : ***Les clignotants ne restent jamais en position allumée, même en cas de coupure***

3.3.2 Programme à vérifier

Afin de vérifier la propriété p_1, le modèle Simulink du *Clignotant* est d'abord traduit[8] en une fonction C, nommée fonction $f1$. La fonction $f1$ possède 6 entrées et 2 sorties. Elle comprend 300 lignes de code, 81 variables booléennes, et 28 variables entières. La fonction $f1$ se compose principalement d'instructions conditionnelles imbriquées et d'instructions d'affectation avec des expressions linéaires ou constantes (cf. partie de code, figure 7).

```
and1_a = (Switch5 && !Unit_Delay3_a_DSTATE);
if(and1_a - Unit_Delay_c_DSTATE != 0)
    rtb_Switch_b = 0;
else{
    add_a = 1 + Unit_Delay1_b_DSTATE;
    rtb_Switch_b = add_a;
}
superior_a = (rtb_Switch_b >= 3);
```

FIG. 7 – Un partie de code de la fonction $f1$

Notre objectif est de vérifier s'il existe une séquence de données d'entrée qui viole la propriété p_1 du *Clignotant*. Cette propriété concerne le comportement du *Clignotant* pour une période de temps infinie. En pratique, on ne peut vérifier qu'une version limitée de la propriété p_1 : nous considérons que la propriété est violée si les feux restent allumés pour une séquence de N unités de temps consécutives. Nous introduisons donc une boucle (bornée par N) qui compte le nombre de fois où la sortie du *Clignotant* a été consécutivement vraie. Si ce compteur est égal à N à la sortie de la boucle, alors la propriété est violée. La partie du programme C correspondant à la version bornée est donnée dans la figure 8.

Le programme tritype et ses spécifications en JML[9], le programme Bsearch, le programme TCAS avec la spécification pour la propriété à vérifier, ainsi que le modèle Simulink détaillé du *Clignotant* sont disponibles à l'adresse `http://users.polytech.unice.fr/~rueher/Publis/jfpc10_fr_annexe.pdf`.

[8] Cette traduction est faite avec un outil propriétaire de Geensys.

[9] cf. `http://www.cs.ucf.edu/leavens/JML/`

```
count = 0;
// number of time where the output has been
// consecutively true int count = 0; consider N
// periods of time
for(int i = 0; i < N; i++) {
// call to f1 function to compute the outputs
// according to non deterministic input values
f1();
if (Model_Outputs4)
   // the output has been consecutively true one
   // more time
   count ++;
else
   // the output is not consecutively true
   count = 0;
}
// if count is less than N, then the property is
// verified
assert(count < N)};
```

FIG. 8 – Le programme C à vérifier

4 Expérimentations et discussion

Dans cette section, nous présentons les expérimentations que nous avons faites pour valider notre approche et nous discutons des travaux futurs.

4.1 Outils

Nous avons comparé les performances de *DPVS* avec CBMC, Euclide et CPBPV*.

Comme indiqué précédemment, CBMC[10] est l'un des meilleurs model checkers bornés actuellement disponibles. Nous avons utilisé la version 3.3.2 qui appelle le solveur *minisat*2.

CPBPV* est une version optimisée de CPBPV [7, 8] qui est implémenté dans `Comet`[11]. Il utilise un ensemble de contraintes pour représenter le programme et ses spécifications, et il explore des chemins exécutables de longueur bornée sur ces ensembles de contraintes. Toutefois, contrairement à CPBPV, il travaille sur un GFC compacté. Une propagation de bornes est également effectuée dans l'étape initiale.

Euclide [14] est un framework de test à base de contraintes conçu pour la génération de données structurales de test, pour la génération de contre-exemple et de preuves partielles des programmes C critiques. Nous n'avons pas pu évaluer Euclide sur l'application *Clignotant*. En effet, en raison d'un bug, Euclide ne pouvait pas traiter ce programme C[12]. Les perfor-

[10] cf. `http://www.cprover.org/cbmc`

[11] `Comet`, une marque commerciale de Dynadec (cf. `http://dynadec.com`) est une plate-forme d'optimisation hybrides.

[12] Nous avons contacté les auteurs, mais ils ne pouvent pas corriger ce bug pour ce moment.

mances d'Euclide sur l'application TCAS sont publiées par les auteurs dans [14]. Les expérimentations des auteurs ont été réalisées sur un processeur Intel Core Duo 2,4GHz avec 2Go de RAM, un ordinateur très semblable à celui que nous avons utilisé.

DPVS est implémenté en `Comet`. Nous donnons ici les expérimentations de deux versions : LIFO et TAS. Nous avons essayé différentes combinaisons de solveurs domaines finis (CP) et solveurs de programmation linéaire (LP). Nous présentons ici les résultats pour deux combinaisons :

- CP-CP : le solveur CP est utilisé pour vérifier la consistance partielle à chaque nœud et pour la recherche d'une solution ;
- LP-CP : le solveur LP est utilisé pour vérifier la consistance partielle d'une relaxation linéaire du système de contraintes à chaque nœud, et le solveur CP est utilisé pour chercher une solution.

Les expérimentations avec $CBMC$,$CPBPV^*$ ont été effectuées sur un processeur Intel Core Duo T8300 2,4 GHz avec 2Go de RAM pour tous les tests sauf l'application *Clignotant* où un Quad Core Intel Xeon X5460 3,16 GHz avec 16Go de mémoire a été utilisé. Tous les temps sont donnés en secondes. OoM signifie "out of memory" tandis que TO signifie "timeout" dans les différents tableaux. TO a été fixé à 3 minutes pour tous les tests.

4.2 Expérimentations sur des exemples académiques

Le tableau 1 fournit les résultats expérimentaux pour le programme *Tritype* et de ses variations. Tous les solveurs ont trouvé un contre-exemple pour les programmes erronés en très peu de temps. Pour les programmes corrects, les preuves sont plus difficiles. En effet, tous les chemins doivent être explorés, contrairement aux versions erronées où la résolution s'arrête dès que la première erreur est trouvée. Toutefois, les systèmes qui utilisent un solveur linéaire ont pu vérifier la correction partielle de Tritype et Triperimeter. La version correcte du programme non-linéaire *Tritimes* est la plus difficile à traiter. Seules les versions de *DPVS* et $CPBPV^*$ qui combinent LP et CP ont pu la traiter. Cela peut s'expliquer par les raisons suivantes :

1. Les décisions sont faciles à tester avec un solveur linéaire ;
2. Les décisions sont ajoutées incrémentalement dans l'ensemble de contraintes ;
3. Le solveur CP qui est appelé à la fin des chemins peut bénéficier des décisions qui ont été ajoutées à l'ensemble de contraintes grâce à un mécanisme de substitution des sous expressions.

Les points (1) et (2) garantissent que des chemins infaisables sont rapidement coupés. Pour expliquer le point (3) considérons la propriété $p : r = i*j*k$ à vérifier. Supposons que la décision $i = j$ a été prise sur un chemin, et donc la valeur retournée par le programme est $i * k * i$. Le système de contraintes contiendra les trois contraintes $r = i*j*k$, $i = j$, $\neg(r = i*k*i)$[13], et le solveur CP peut facilement détecter l'inconsistance.

Le tableau 2 donne les résultats expérimentaux sur une version correcte du programme de *recherche binaire*. CBMC et *DPVS* ne peuvent pas traiter ce benchmark. CBMC perd beaucoup de temps dans le processus de dépliage. Les stratégies utilisées par *DPVS* ne sont pas bien adaptées non plus pour ce programme très spécifique. La version LP-CP de CPBPV* est par contre très efficace. Elle utilise une approche descendante et ajoute incrémentalement les décisions prises sur un chemin. Cette stratégie est particulièrement bien adaptée pour ce programme de *recherche binaire* qui a une pre-condition très contraignante.

Tab. 1 – Tritype (Typ), Tritimes (Tm), Triperimeter (per) , entiers de 16 bits

Prog	CBMC 3.3.2	DPVS LIFO CP-CP	CPBPV* CP-CP	DPVS LIFO LP-CP	CPBPV* LP-CP
Typ-OK	0.161	TO	TO	0.781	0.304
Typ-KO	0.012	0.087	0.127	0.030	0.009
Per-OK	0.717	TO	TO	0.054	0.054
Per-KO	0.015	0.002	0.027	0.014	0.018
Tm-OK	TO	TO	TO	1.048	0.865
Tm-KO	0.050	0.070	0.002	0.029	0.009

Tab. 2 – Cherche binaire (entiers de 16 bits)

Length	CBMC 3.3.2 (MiniSAT2)	DPVS LIFO LP-CP	CPBPV* LP-CP
4	5.732	0.529	0.107
8	110.081	35.074	0.298
16	TO	TO	1.149
32	TO	TO	5.357
64	TO	TO	27.714
128	TO	TO	153.646

4.3 Expérimentations sur TCAS

Les résultats pour l'application TCAS sont présentés dans le tableau 3. CBMC a de bons résultats sur ce problème, mais les performances de la combinaison CP-CP de *DPVS* sont meilleures. Ceci s'explique par le fait que TCAS est essentiellement un problème SAT : le programme contient de nombreux booléens

[13] Nous ne détaillons pas ici les renommages DSA.

et les entiers ne peuvent prendre que quelques valeurs. C'est aussi pourquoi la combinaison CP-CP de CPBPV* est meilleure que la combinaison de LP-CP CPBPV* sur ce problème.

Les différentes stratégies de *DPVS* donnent des résultats semblables sur ce problème. Les expérimentations sur *TCAS* ont été réalisées pour des entiers de 16 bits car nous voulions comparer nos résultats à ceux publiés dans [15]. Euclide est beaucoup plus lent que les autres solveurs, mais il est implémenté en Prolog.

TAB. 3 – TCAS (entiers de 16 bits)

Property	CBMC 3.3.2	Euclide (SICtus Prolog)	DPVS LIFO CP-CP	CPBPV* CP-CP	CPBPV* LP-CP
P1A-OK	0.101	0.7	0.021	0.173	8.960
P1B-OK	0.063	0.7	0.019	0.211	13.737
P2A-OK	0.097	0.6	0.020	0.175	9.168
P2B-KO	0.064	0.7	0.011	0.014	0.165
P3A-KO	0.061	5.4	0.008	0.036	0.615
P3B-OK	0.066	1.2	0.011	0.100	0.4431
P4A-KO	0.075	6.8	0.008	0.045	1.446
P4B-KO	0.060	2.7	0.008	0.014	0.254
P5A-OK	0.068	0.6	0.044	0.197	20.627
P5B-KO	0.065	1.0	0.015	0.014	0.195

4.4 Expérimentations sur l'application Clignotant

TAB. 4 – *Clignotant*

N	CBMC 3.3.2	DPVS HEAP CP-CP	DPVS LIFO CP-CP	DPVS HEAP LP-CP	DPVS LIFO LP-CP
5	0.134	0.026	0.032	0.565	0.953
10	0.447	0.055	0.068	2.484	4.134
20	1.766	0.118	0.149	9.081	15.320
30	4.231	0.194	0.248	20.847	35.066
50	12.92	0.345	0.458	57.797	96.693
75	32.747	0.602	0.842	137.104	TO
100	58.279	2.750	3.394	TO	TO
150	138.192	1.552	2.365	TO	TO
200	OoM	OoM	8.082	TO	TO

CBMC et *DPVS* ont trouvé des contre-exemples qui violent la propriété p_1. Ils ont également généré des données d'entrée pour lesquelles le clignotant reste allumé de manière continue pour une séquence de N unités de temps.

Par exemple, voici une séquence d'entrées qui viole la propriété p_1 pour une séquence de 5 unités de temps :

```
[(0,1,0,0,0,1),(0,1,0,0,0,1),(0,1,0,0,0,1),
(0,0,1,0,0,1),(0,0,0,0,0,1)]
```

où $(0, 1, 0, 0, 0, 1)$ sont les valeurs des entrées de in_1 à in_6 pour la première période, $(0, 1, 0, 0, 0, 1)$ les valeurs des entrées pour la deuxième période et ainsi de suite.

Le tableau 4 montre que la combinaison CP-CP de *DPVS* est bien meilleure que les combinaisons LP - CP sur l'application *Clignotant*. On notera que par manque de mémoire, CBMC n'est pas capable de générer des contre exemples pour plus de 200 unités de temps alors que *DPVS*, avec la stratégie TAS et la combinaison CP-CP, peut générer des contre-exemples pour 400 unités de temps. CPBPV* n'est pas capable de générer des contre-exemples pour plus de 10 unités de temps.

L'utilisation d'un solveur de contraintes sur les domaines finis pour calculer la consistance partielle est beaucoup efficace sur ce programme qu'un solveur de programmation linéaire. Ceci s'explique pour deux raisons :

- la relaxation linéaire nécessaire par le solveur LP introduit de nombreux points de choix alors que ceci n'est pas le cas pour le solveur CP. En effet, considérons une condition telle que $x == y$ dans un `if`. La négation de cette condition correspond à la contrainte $x! = y$ qui nécessite la génération de deux systèmes de contraintes linéaires : l'un avec la contrainte $x < y$, l'autre avec la contrainte $x > y$.
- les domaines des variables entières sont petits pour cette application, et l'étape de propagation des bornes que nous effectuons avant l'exploration du graphe permet de réduire fortement la taille de ces domaines. Les tests de consistance avec le solveur CP sont donc très efficaces.

4.5 Discussion et travaux futurs

Les premières expérimentations avec *DPVS* sont très encourageantes. *DPVS* se comporte bien sur des exemples académiques et obtient de bien meilleurs résultats que CBMC sur deux applications réelles. Il faut souligner ici que la génération de cas de test pour des séquences de périodes de temps assez grandes est un problème crucial dans les applications temps-réel.

Ces résultats sont très prometteurs car *DPVS* est encore un prototype académique tandis que CBMC est un des tout meilleurs systèmes de model checking actuellement disponibles. *DPVS* est également bien meilleur que Euclide sur l'application TCAS. Naturellement, d'autres expérimentations sur des applications réelles sont nécessaires pour affiner et pour valider l'approche proposée. Il nous faut en particulier arriver à mieux expliciter les caractéristiques des programmes pour lesquels cette stratégie est effectivement bien adaptée.

Les travaux futurs concernent l'extension de notre prototype. Actuellement, il y a beaucoup de restrictions sur les programmes C et Java que le prototype peut traiter. En particulier, nous ne traitons que

des programmes qui ne provoquent pas un déroutement, e.g., nous ne traitons pas les programmes qui contiennent des divisions par zéro ou des erreurs qui provoquent la levée d'une exception. Nous acceptons des tableaux de taille bornée, mais le type des données d'entrée est limité aux booléens et aux entiers. Nous travaillons aussi actuellement sur l'interface entre un solveur CP et notre propre solveur sur les nombre à virgule flottante [4].

Références

[1] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *LOPSTR 2008*, LNCS (Springer verlag), vol. 5438 : 4–23. Springer, 2008.

[2] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs In *Information Processing Letters*, vol. 93(6) :281–288.

[3] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Model checking flight control systems : The Airbus experience. In *ICSE 2009*, pages 18–27. IEEE, 2009.

[4] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2) :97–121, 2006.

[5] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying safety-critical systems. In *ESEC / SIGSOFT FSE*, pages 142–151, 2001.

[6] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, LNCS (Springer verlag),vol. 3920 :182–196, 2006.

[7] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. In *CP 2008*, LNCS (Springer verlag), vol. 5202 :327–341, 2008.

[8] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. *Constraints Journal*, Springer Verlag, vol. 15(2) :238-264, 2010.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, 1991.

[10] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. Improving constraint-based testing with dynamic linear relaxations. In *Proc. of ISSRE*, pages 181–190. IEEE Computer Society, 2006.

[11] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008.

[12] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers : a survey. *Softw. Test., Verif. Reliab.*, 19(3) :215–261, 2009.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart : directed automated random testing. In *PLDI*, pages 213–223. ACM Press, 2005.

[14] Arnaud Gotlieb. Euclide : A Constraint-Based Testing Framework for Critical C Programs. In *ICST'09*, pages 151–160. IEEE Computer Society, 2009.

[15] Arnaud Gotlieb. TCAS software verification using constraint programming. *The Knowledge Engineering Review, (Accepted for publication)*, 2010.

[16] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.

[17] Paula Herber, Florian Friedemann, and Sabine Glesner. Combining model checking and testing in a continuous hw/sw co-verification process. In *TAP 2009*, LNCS (Springer verlag), vol. 5668 : 121–136, 2009.

[18] Daniel Jackson and Mandana Vazir. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25. ACM Press, 2000.

[19] Carolos Livadas, John Lygeros, and Nancy A. Lynch. High-Level Modeling and Analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[20] N.Williams, B.Marre, P.Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing -EDCC'05*, pages 281–292, 2005.

[21] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *ASE*, pages 13–21. IEEE Computer Society, 2001.