CP-SST : approche basée sur la programmation par contraintes pour le test structurel du logiciel

Abdelilah Sakti **Yann-Gaël Guéhéneuc** **Gilles Pesant**

Département de Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
{Abdelilah.Sakti, Yann-Gael.Gueheneuc, Gilles.Pesant}@polymtl.ca

Résumé

Le coût du test peut facilement dépasser 50% du coût total d'un logiciel critique. Le test structurel est la stratégie de choix pour tester un système critique. En fonction de la criticité du système, différentes applications de test structurel sont exigées (analyse de couverture structurelle, génération des données de test, preuve de la post condition). Cet article s'intéresse au problème de combinaison des applications de test structurel, peu touché jusqu'à maintenant, qui peut faciliter l'automatisation du processus de test structurel et réduire de manière significative le nombre de données de test générées. Pour intégrer plusieurs applications de test structurel en une seule approche, nous modélisons le programme sous test et son graphe de flot et de contrôle (GFC) par un problème de satisfaction de contraintes (PSC). Nous utilisons une nouvelle classification des sommets du GFC, la dépendance de contrôle, les techniques statiques d'assignation unique (SSA) et l'inférence de la programmation par contraintes. Le modèle PSC que nous proposons conserve toute la sémantique structurelle du programme, cette caractéristique le rendant utilisable pour différentes applications de test structurel : analyser une couverture structurelle, générer des données de test ou prouver la post condition. Nos expérimentations sur des benchmarks traditionnels montrent un gain de temps par rapport aux approches existanes de test structurel.

1 Introduction

Le test du logiciel est une étape importante dans le cycle de vie d'un logiciel. son coût peut facilement dépasser 50% du coût total d'un logiciel critique [16]. Les principales raisons qui obligent les fabricants de systèmes critiques à tester leurs produits sont les coûts énormes et les dommages qui peuvent être causés par un bogue logiciel : un comportement indésirable du système peut mener à une catastrophe. Un exemple célèbre est survenu en 1996, quand un bogue dans le composant de référence inertiel de la fusée a causé la destruction d'Ariane 5 juste 40s après le lancement. L'Agence spatiale européenne a annoncé un retard du projet, a estimé une perte économique directe de 500 millions de dollars US [1] et, implicitement, une perte de confiance des clients aux avantages de la concurrence. Ce genre de bogue est difficile à accepter parce qu'il conduit à des pertes financières énormes. Il pourrait également avoir des effets négatifs sur l'environnement ou mettre la vie ou la santé humaine en danger. Dans la catégorie des systèmes critiques, la sécurité humaine ou environnementale est largement basée sur les fonctionnalités du système. Afin d'augmenter la confiance en ces systèmes, différentes normes existantes (i.e., DO-178B pour l'aéronautique, la CEI 61513 pour le nucléaire, la CEI 50126 pour le ferroviaire). Une norme exige un niveau minimal de couverture de test pour une fonctionnalité du système en fonction de sa criticité. Pour la norme DO-178B [17], les programmes sont classés sur cinq niveaux (de E à A), chaque niveau représente l'effet possible d'une erreur du programme sur la sécurité des passagers. Le critère de couverture d'un programme est défini selon son classement dans la norme [14, 17] : un programme de niveau C doit être testé par un jeu de cas de test qui permette une couverture de toutes les instructions. Un programme de niveau B doit être testé par un jeu de données test qui permette une couverture de toutes les décisions. Un programme de niveau A doit être testé

par un jeu de cas de test qui permette une couverture de toutes les décisions conditions modifiées. Pour cette norme, tout cas de test doit être généré à partir des spécifications, ce qui veut dire que la principale tâche du test structurel est de mesurer la couverture des données de test, de montrer les parties du programme qui ne sont pas explorées et, dans certains cas, de générer le jeu de données de test complémentaire. Cette norme montre qu'un même programme au niveau A ou un système qui contient des programmes qui sont classés dans différents niveaux ont besoin d'une combinaison de plusieurs critères de couverture et plusieurs applications de test structurel (analyse ou génération de données de test).

Les principales applications de test structurel sont : l'analyse de la couverture structurelle, la génération des données de test pour satisfaire un critère de couverture ou atteindre un point donné dans un programme (i.e., générer une exception de division par zéro) et la preuve de post condition ou la génération d'un contre exemple. Généralement, le test d'un système critique exige une combinaison de ces applications et pour la certification, il est fortement recommandé de travailler sur un même outil de test [11]. Il est donc avantageux de combiner plusieurs applications de test structurel en une seule approche générique pour répondre aux diverses exigences de test structurel. Regrouper alors toutes les applications citées précédemment en une seule approche représente un défi très intéressant.

Dans cet article, nous proposons une nouvelle approche qui permet d'analyser la couverture structurelle, de générer des données de test structurel et de prouver la post condition ou de générer un contre exemple. Nous utilisons un problème de satisfaction de contraintes (PSC) qui modélise le programme sous test (PST) et son graphe de flot et de contrôle (GFC). L'approche est basée sur de nouveaux mécanismes qui utilisent la forme SSA [5, 8] et les dépendances de contrôle [8, 9]. L'utilisation de ces deux derniers mécanismes n'est pas nouvelle : ce qui est original dans ce travail est l'utilisation de la programmation par contraintes (PC) pour analyser une couverture structurelle, le regroupement des principales applications du test structurel en une seule approche et la modélisation du GFC par un PSC.

La suite de cet article est organisée comme suit : À la section 2, nous discutons les travaux connexes. À la section 3, nous introduisons les définitions et les concepts nécessaires pour l'approche. Les principes et les mécanismes utilisés pour concevoir un PSC sont expliqués à la section 4. La section 5 présente les résultats expérimentaux, en comparant notre approche à l'état de l'art et la section 6 présente des conclusions et des travaux futurs.

2 Travaux connexes

Dans la littérature, il existe des travaux qui sont liés à notre approche par l'utilisation de la PC pour répondre aux besoins d'un sous-ensemble d'applications du test structurel. La plupart de ces travaux sont orientés chemins [2, 18]. La technique orientée chemins consiste à extraire un chemin à partir de GFC ou d'une version instrumentée du PST, puis de générer un prédicat de ce chemin. La résolution de ce prédicat génère des données de test correspondant au chemin choisi. Cette procédure est répétée pour chaque élément (chemin, branche) de l'objectif de test (tous les chemins, toutes les branches). Nous signalons deux faiblesses dans cette technique : d'abord, le processus de génération de prédicat a un coût important lié à l'évaluation symbolique, en deuxième lieu, en général, cette technique ne peut pas générer des cas de test pour atteindre un point donné dans un PST. PathCrawler [18] et Osmose [2] sont deux outils qui peuvent générer un jeu de données de test pour une couverture de tous les chemins ou les k-chemins (PathCrawler pour les programmes en C, Osmose pour les exécutables). Ces approches peuvent être appliquées aux autres langages de programmation mais elles ne permettent pas d'analyser une couverture structurelle, de générer des données de test pour d'autres critères de couverture ou de prouver la post-condition.

Il existe aussi des approches orientées buts [4, 6, 12, 11] qui génèrent des données de test pour atteindre une instruction, une branche ou un chemin bien déterminé. En règle générale, ces approches traduisent un PST en un PSC en passant par un modèle SSA pour éviter le coût de l'évaluation symbolique. Contrairement aux approches orientées chemins, cette technique identifie difficillement des points du PST nécessaires pour atteindre un critère de couverture donné. Collavizza et al. [6] propose un cadre de travail pour prouver la post-condition d'un PST en Java ; une version améliorée a été implémentée dans [7], cette approche peut être utilisée pour prouver la satisfaction de la post-condition pour différents langages de programmation mais elle ne permet pas d'analyser une couverture structurelle ni de générer des données de test. INKA [4, 12] est un outil pionnier qui utilise la PC pour générer des données de test pour les programmes C. Les auteurs d'INKA proposent une approche qui peut être utilisée indépendamment du langage de programmation mais cette approche ne permet pas d'analyser une couverture structurelle ni de générer des données de test pour certains types de critères de couverture structurelle (i.e., la couverture tous les chemins, decision/condition modifiée). Euclide [11] est le seul outil basé sur la PC et orienté buts qui regroupe trois applications du test

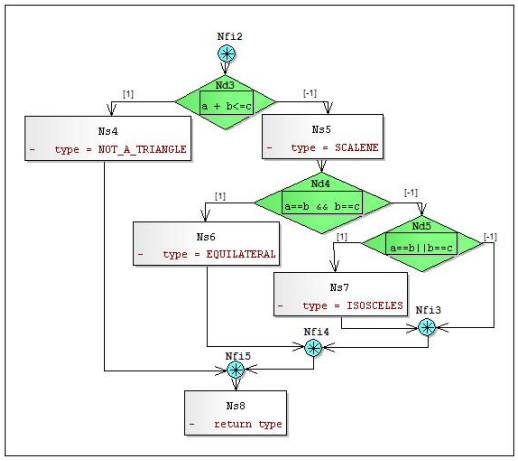

FIGURE 1 – Une partie du GFC de la fonction *tri_type* (lignes 13 à 29).

structurel. Il permet de prouver la post-condition et de générer les données de test pour quelques critères de couverture : toutes les instructions et toutes les décisions, mais avec une identification explicite des points à atteindre dans le code. Cet outil ne permet pas l'analyse d'une couverture structurelle et ne permet pas de générer des données de test pour d'autre mèsures de couverture tel que tous les k-chemins.

Notre approche combine les deux techniques, on peut dire qu'elle est orientée but et chemin. En utilisant la forme SSA, notre approche ne souffre pas du coût de l'évaluation symbolique ni de l'identification des points requis pour un critère de couverture donné, qui est résolu par la modélisation du GFC. En outre, elle est applicable pour répondre aux besoins variés du test structurel : analyser une couverture structurelle, générer de données de test pour atteindre un point dans le PST ou satisfaire un critère de couverture et de prouver la post-condition ou générer un contre-exemple.

3 Notions de base

Le programme triangle, utilisé dans des travaux antérieurs [2, 4, 15, 18], permet à l'utilisateur d'identifier un triangle par la longueur de ses trois côtés. Le code du Prog.1 est une implémentation possible de ce programme [15]. Cette implémentation est différente de celle utilisée dans [4, 6, 18]. Nous utilisons le programme triangle pour exprimer, définir et illustrer notre approche.

Definition 1. *Dans un graphe de flot et de contrôle (GFC) [9], nous distinguons cinq classes de noeuds : noeud de décision, noeud de jointure, noeud d'instruction et noeud des paramètres et variables globales.* Un noeud de décision *est un noeud qui a deux ou plusieurs arêtes sortantes. Il représente une instruction de contrôle. Sur la Fig. 1* $Nd3$, $Nd4$ *et* $Nd5$ *sont des noeuds de décision.* Un noeud de jointure *est un noeud qui a deux ou plusieurs arêtes entrantes. Il représente la fin d'une instruction de contrôle. Sur la Fig. 1* $Nfi2$, $Nfi3$, $Nfi4$ *et* $Nfi5$ *sont des noeuds de jointure.* Un noeud de condition *est un noeud dérivé d'un noeud de décision. Si une expression logique d'un noeud de décision est composée de plusieurs conditions, alors la décomposition de cette expression à des conditions atomiques génère un ensemble de ces noeuds. Sur la Fig. 1, cette classe de noeuds est cachée dans les décisions* $Nd4$ *et* $Nd5$. Un noeud d'instruction *est un noeud qui accepte une seule arête entrante et une autre sortante. Il représente une séquence d'instructions qui ne contient aucune instruction de contrôle. Sur la Fig. 1,* $Ns4$, $Ns5$, $Ns6$, $Ns7$ *et* $Ns8$ *sont des noeuds d'instruction.* Une décision positive *(resp.* une décision négative*) est une branche qui satisfait (resp. ne satisfait pas) la condition d'un noeud de décision. Dans notre exemple, la Fig.1, l' arête de* $Nd3$ *à* $Ns4$ *est une décision positive et celle de* $Nd3$ *à* $Ns5$ *est une décision négative. Nous utilisons le signe* "$+$" *pour désigner la décision positive et le signe* "$-$" *pour la décision négative.* Un noeud actif *est un noeud*

Prog. 1 le code source de la fonction *tri_type* en language C.

```
01 int tri_type(int a, int b, int c)  02 {
03    int type;
04    int t;
05    if (a > b){
06         t = a; a = b; b = t;       07 }
08    if (a > c){
09         t = a; a = c; c = t;       10 }
11    if (b > c){
12         t = b; b = c; c = t;       13 }
14    if (a + b <= c){
15          type = 0;     16 }
17    else 18 {
19         type = 1;
20         if (a == b && b == c){
21              type = 2; 22 }
23         else {
24              if (a == b || b == c){
25                   type = 3; 26 }
27         } 28 }
29    return type;
30 }
```

qui fait partie du chemin d'exécution sélectionné, tout noeud qui n'appartient pas au chemin sélectionné est appelé noeud neutre.

Definition 2. Une dépendance de contrôle *[8] est une relation entre deux noeuds $N1$ et $N2$. $N2$ est en dépendance de contrôle avec $N1$ si et seulement si l'exécution de $N2$ dépend du résultat de l'exécution du noeud $N1$. Par exemple, sur la Fig. 1, $Ns4$ est en dépendance de contrôle avec $Nd3$.*

Definition 3. Une contrainte gardée *[13], également appelée contrainte conditionnelle, est une contrainte composée de deux parties : une garde et un objectif. L'ajout de l'objectif au PSC dépend de l'état de la garde : si la garde est impliquée (satisfaite) par l'état du PSC, alors l'objectif est inséré dans le PSC et la contrainte gardée est enlevée ; si la négation de la garde est impliquée par l'état du PSC, alors la contrainte gardée est retirée de PSC ; si le solveur ne peut pas prouver l'une ou l'autre, il met la contrainte gardée en état de suspension.*

Definition 4. La forme SSA *[5, 8] est une représentation formelle d'un programme, elle conserve la sémantique du programme. Elle définit les variables de façon unique à chaque instruction d'affectation et chaque utilisation peut être atteinte depuis cette définition. Par exemple, pour une variable V, une nouvelle variable dérivée V_i est générée à chaque nouvelle définition de V rencontrée dans le programme. La variable V est remplacée par V_i dans toute instruction qui utilise V jusqu'à la définition suivante de V. Ce processus de renommage est géré par une fonction σ : pour simplifier, $\sigma(V)$ renvoie le nombre d'instructions d'affectation de V qui précédent l'instruction appelante. Dans un programme qui contient plusieurs branches, les variables dérivées d'une même variable peuvent atteindre le même point de jointure. Ces variables dérivées doivent être fusionnées en une seule variable qui est utilisée à partir de la jointure. Cette variable est générée par une fonction φ : si une instruction de contrôle atteint son noeud de joiture via sa branche numéro i, la valeur retournée par cette fonction est celle de l'opérande numéro i. Une fonction φ est ajoutée à la fin de chaque instruction de contrôle.*

4 Approche

Les classes des noeuds et les relations de dépendance sont la base de notre approche pour modéliser un PST par un PSC. Un PST est traduit en un ensemble de contraintes dont la majorité sont des contraintes gardées. Ces contraintes sont obtenues par la traduction d'une instance de relation entre deux noeuds ou entre un noeud et une instruction. Les noeuds du GFC sont représentés par des variables entières avec un petit domaine $\{-1, 0, 1\}$: -1 signifie que le noeud est actif et sa décision est négative, 0 signifie que le noeud est neutre ; 1 signifie que le noeud est actif et sa décision est positive. Pour répondre à un besoin de test structurel d'un PST en utilisant notre approche, nous proposons quatre étapes principales : le modèle SSA ; la modélisation par contraintes du GFC ; la construction de PSC global ; et la résolution du PSC.

Mod. 1 Modèle SSA de la fonction *tri_type*.

```
Np0 int tri_type(int a0, int b0, int c0){
Ns0     int type0=0;
Ns0     int t0=0 ;
Nd0     if (a0 > b0){
Ns1           t1 = a0; a1 = b0; b1 = t1; }
Nfi0    t2=fi(t1,t0); a2=fi(a1,a0);
Nfi0            b2=fi(b1,b0);
Nd1     if (a2 > c0){
Ns2           t3 = a2; a3 = c0; c1 = t3; }
Nfi1    t4=fi(t3,t2); a4=fi(a3,a2);
Nfi1            c2=fi(c1,c0);
Nd2     if (b2 > c2){
Ns3           t5 = b2; b3 = c2; c3 = t5; }
Nfi2    t6=fi(t5,t4); b4=fi(b3,b2);
Nfi2            c4=fi(b3,b2);
Nd3     if (a4 + b4 <= c4)
Ns4          type1 = 0;
        else{
Ns5          type2 = 1;
Nd4          if (a4 == b4 && b4 == c4)
Ns6               type3 = 2;
             else{
Nd5               if (a4 == b4 || b4 == c4)
Ns7                    type4 = 3;
Nfi5              type5=fi(type4, type2);
             }
Nfi4       type6=fi(type3,type5);
        }
Nfi3    type7=fi(type1,type6);
Ns8     return type7;
    }
```

4.1 Le modèle SSA

Si un PST contient deux instructions d'assignation d'une même variable à deux valeurs différentes (i.e., $x = 1$ et $x = 2$), la traduction directe de ces deux instructions en contraintes génère deux contraintes incohérentes. Pour éviter ce problème bien connu

[6, 12, 11], avant de traduire le PST en un PSC, on le traduit en un modèle SSA. Le Mod. 1 montre le modèle SSA de la fonction de *tri_type*.

4.2 La modélisation par contraintes du GFC

Cette étape consiste à modéliser le GFC du PST par un PSC préliminaire. Nous commençons par la génération d'un GFC qui est caractérisé par des noeuds indépendants pour chaque classe. Puis, nous identifions les noeuds du GFC selon leur classe d'appartenance et leur ordre dans cette classe : un noeud de décision est identifié par un préfixe Nd et un indice i ; un noeud de condition est identifié par un préfixe correspondant au noeud de décision d'origine Ndi et un indice j ; un noeud d'instructions est identifié par un préfixe Ns et un indice i. Sur le Mod. 1 à la gauche de chaque instruction, nous mentionnons le noeud correspondant du GFC. Après l'identification des noeuds, nous étiquetons les arrêtes selon leur noeud d'origine : une arrête sortante d'un noeud d'instruction est étiquetée par 1 ; une arête sortante d'un noeud de condition ou de décision est étiquetée par 1 si la décision est positive ou par -1 si la décision est négative.

TABLE 1 – Noeud-Noeud : Table de contraintes

Nd_i	N_j **dans** Nd_i-	N_k **dans** Nd_i+
0	0	0
1	0	-1
1	0	1
-1	-1	0
-1	1	0

TABLE 2 – Décision conjonctive : Table de contraintes

Nd_i	Nd_{i1}	...	Nd_{ij}	...	Nd_{in}
1	1	1	1	1	1
0	0	0	0	0	0
-1	-1	*	*	*	*
...	...	...	...	...	...
-1	*	*	-1	*	*
...	...	...	...	...	...
-1	*	*	*	*	-1

$Noeud - Noeud$ est une relation exprimée par deux noeuds, dans laquelle au moins l'un des deux est un noeud de décision. Lorsqu'un noeud est dans l'une des branches d'un noeud de décision, le premier ne peut être neutre si cette branche est active. Cette relation est alors exprimée par les contraintes illustrées dans la Tab. 1. La relation $Noeud - Noeud$ est la règle principale pour obtenir le modèle à contraintes d'un GFC.

Afin de montrer les branches cachées d'un noeud de décision multi-conditions, à partir de ce noeud de décision, nous dérivons un graphe de décisions en décomposant l'expression multi-conditions en conditions atomiques. Une décision multi-conditions utilise deux formes d'expressions logiques : la forme conjonctive ou la forme disjonctive. Chaque forme peut être exprimée en termes de ses conditions atomiques : Une forme conjonctive (resp. disjonctive) est vraie (resp. fausse) si et seulement si toutes ses conditions atomiques sont vraies (resp. fausses). La relation entre un noeud de décision conjonctif Nd_i et ses noeuds de conditions dérivés Nd_{ij} est exprimée par les contraintes dans la Tab. 2. La relation entre un noeud décision disjonctif Nd_i et ses noeuds de conditions dérivés Nd_{ij} est exprimée de façon similaire à la forme conjonctive. Pour obtenir le tableau des contraintes d'une décomposition disjonctive, il suffit de permuter les 1 et les -1 dans la Tab. 2.

Pour les expressions complexes qui combinent des formes conjonctives et disjonctives, il est préférable d'utiliser des noeuds intermédiaires. La décomposition des noeuds de décision n'est pas nécessaire si le besoin de test n'exige pas le niveau de condition.

Lors de la génération de PSC préliminaire, un noeud est traduit par une variable dont le domaine est l'ensemble d'étiquettes de ses arêtes sortantes, sauf pour les noeuds de jointure qui prennent le domaine du noeud de décision correspondant. Si un noeud n'est pas la racine, on ajoute la valeur 0 à son domaine. Nous utilisons les règles de décomposition et la règle $Noeud - Noeud$ pour générer des contraintes qui expriment les relations entre les noeuds. Le modèle Mod.2 montre le PSC résultat de la modélisation du GFC de la fonction *tri_type* : la première partie (01-04) contient les définitions des variables et de leurs domaines ; la deuxième partie (05-12) exprime les relations entre noeuds du GFC, elle a été générée selon la règle $Noeud - Noeud$; la troisième partie (13-16) exprime la relation entre les décisions et leurs noeuds de conditions dérivés. Elle a été générée en utilisant les règles de décomposition.

4.3 Construction du PSC global

La troisième étape utilise le PSC préliminaire, le modèle SSA et la relation entre un noeud et les instructions qu'il représente pour créer un nouveau PSC global. Elle consiste à traduire chaque instruction de modèle SSA en une ou plusieurs contraintes. Dans cette section, nous discutons aussi certaines particularités des boucles et des expressions d'affectation et de déclaration des variables.

$Noeud - Instruction$ est une relation exprimée sur un noeud d'instruction et son contenu. Une instruction peut être considérée comme une contrainte qui est nécessairement satisfaite si son noeud est actif. Il

Mod. 2 PSC du GFC de la fonction *tri_type*.

```
01 X{Nsm , Ndj, Nfij, Nd4k, Nd5k
          / 0<=m<=8, 0<=j<=5, 0<=k<=2};
02 D(Ns0)=D(Ns8)={1}; D(Nsi / 0<i<=7)={0,1};
03 D(Ndj)=D(Nfij)={-1,1} / 0<=j<=3;
04 D(Ndj)=D(Nfij)=D(Ndjk)= {-1,0,1}
          / 2<=j<=5, 0<=k<=1;
05 C{ //les contraintes suivantes sont
     //générées selon la règle Noeud-Noeud
06      table (Nd0,Ns1, Tab.1);
07      table (Nd1,Ns2, Tab.1);
08      table (Nd2,Ns3, Tab.1);
09      table (Nd3,Nd4, Tab.1);
10      table (Nd4,Ns6, Tab.1);
11      table (Nd5,Ns7, Tab.1);
12      Nd0=Nfi0; Nd1=Nfi1; Nd2=Nfi2;
       Nd3=Nfi3; Nd4=Nfi4; Nd5=Nfi5;
13 //les contraintes suivantes sont générées
    // selon la règle de décomposition
14      table (Nd4, Nd40, Nd41, Tab.2);
15      table (Nd5, Nd50, Nd51, Tab.2);
16 }
```

existe une relation entre un noeud d'instruction et son contenu : les instructions qu'il représente. Si un noeud Ns_i est actif (prend la valeur 1), toute instruction S de ce noeud est exécutée. $Noeud-Condition$ est une relation exprimée sur un noeud de décision et son instruction de contrôle : si le noeud est actif, alors sa condition prend une valeur vrai ou faux selon la décision prise. Il existe une relation d'implication entre un noeud de décision Nd et la condition de son instruction de contrôle C. Pour exprimer les relations entre un noeud de GFC et son contenu d'instructions, nous utilisons les règles $Noeud - Instruction$ et $Noeud - Condition$. Chaque instruction est traduite en au moins une contrainte : une définition d'une variable dans le noeud paramètre se traduit par la définition de la variable correspondante dans le PSC. Dans un noeud d'instruction Ns_i, une instruction d'affectation S (i.e., $type_2 = 1$) se traduit par une contrainte gardée dont la garde est une expression logique d'égalité entre la variable Ns_i et la valeur 1 (i.e., $Ns_5 = 1 \rightarrow type_2 = 1$). Si un noeud d'instruction est neutre ($Ns_i = 0$), une variable qui est définie dans ce noeud peut forcer le solveur à générer un nombre important de solutions symétriques. Pour éviter ce problème, nous affecttons à cette variable sa dernière valeur avant l'instruction de contrôle parent(i.e., $Ns_5 = 0 \Rightarrow type_2 = type_0$). Dans un noeud de décision ou de condition Nd_i, la condition de son instruction de contrôle C (i.e., $a_4 + b_4 < c_4$) se traduit par deux contraintes gardées, la première (resp. la seconde) utilise C (resp. négation de C) comme partie objectif et l'expression correspond à la décision positive (resp. négative) du noeud en tant que garde (i.e., $Nd_3 = 1 \Rightarrow b_4 + a_4 < c_4; Nd_3 = -1 \Rightarrow not(a_4 + b_4 < c_4)$). Dans un noeud de jointure, une fonction φ se traduit par une contrainte gardée pour chaque valeur dans son domaine (i.e., $Nfi_4 = 1 \Rightarrow type_6 = type_3; Nfi_4 = -1 \Rightarrow type_6 = type_5; Nfi_4 = 0 \Rightarrow type_6 = type_2$).

Déclaration des variables

Une déclaration de variable V_0 de type $T(entier, caractere, booleen)$ se traduit dans le PSC global en une nouvelle variable V_0 de type T. Le domaine de V_0 est extrait par l'analyse du PST, il peut être borné entre deux valeurs V_{min} et V_{max}, sinon le domaine contient toutes les valeurs de T supportées par le système (i.e., sur un système 32-bit, de type entier est définie dans l'intervalle $[-2^{31}, 2^{31} - 1]$).

Un tableau de taille fixe est traduit dans le PSC par un tableau de même taille. Une déclaration d'un tableau t_0 de taille variable (allocation dynamique) se traduit dans PSC par un tableau de taille prédéfinie l_{max} et une variable l_{t0} qui représente sa taille réelle. Pour ignorer les indices qui sont supérieurs à l_{t0}, nous ajoutons quelques contraintes : ces variables sont fixées à une constante et sont ignorées par toute autre contrainte sur le tableau.

Instructions et expressions d'affectation

Nous distinguons deux types d'instructions d'affectation : celles qui sont exprimés sur des variables scalaires, et celles qui sont exprimées sur des variables de type tableau.

1. Sur une variable scalaire : l'instruction se traduit par une déclaration d'une nouvelle variable et une contrainte gardée conformément à la règle $Noeud - Instruction$.

2. Sur une variable tableau : dans un programme, un tableau est manipulé par deux groupes d'instructions : lecture et écriture. La lecture d'un élément ne pose aucun problème, car l'instruction est traduite de la même manière qu'une instruction simple, c'est-à-dire que la variable du tableau est remplacée par la variable équivalente. Dans le cas d'une écriture d'un élément, une nouvelle variable du tableau est générée qui contient les mêmes éléments que son prédécesseur sauf l'élément qui contient une nouvelle donnée. Par exemple, l'af-

fectation $t[i] = x$ est traduite selon la formule :

$$T_{\sigma(T)}[k] = \begin{cases} x_{\sigma(x)} & \text{si } k = i \\ T_{\sigma(T)-1}[k] & \text{sinon} \end{cases}$$

Boucles
Toute boucle du PST doit être transformée en une boucle $tant - que$ ($While$). Nous utilisons une constante k ($k - chemin$) afin de limiter le nombre d'itérations dans une boucle. Avec cette limitation, une boucle contient $k + 1$ noeuds de décision. Pour forcer le PST à quitter la boucle au maximum après k itérations, le dernier noeud de décision Nd_{k+1} doit toujours être différent de la valeur 1. Le résultat de la traduction d'une boucle ($While\ C\ B$;) est :

1. $Nd_0 \neq 0$;
2. $\forall\ 0 < i \leq k+1,\ Nd_i \neq 0 \Leftrightarrow Nd_{i-1} = 1$;
3. $\forall\ 0 \leq i \leq k+1$
   - $Nd_i = 1 \Leftrightarrow Ns_i = 1$, où Ns_i est le noeud d'instruction qui contient B à la i^{eme} itération ;
   - $Nd_i = Nfi_i$, où Nfi_i est le noeud de jointure correspond au noeud de décision Nd_i ;
   - $Nd_i = 1 \Rightarrow C_{\sigma i(C)}$, où $C_{\sigma i(C)}$ est la forme SSA de C à la i^{eme} itération ;
   - $Nd_i = -1 \Rightarrow not(C_{\sigma i(C)})$;
4. $Nd_{k+1} \neq 1$.

4.4 Résolution du PSC

La méthode de traduction proposée donne un PSC qui maintient toute la sémantique structurelle du PST. Cette propriété rend notre approche capable de répondre aux différents besoins de test structurel : analyser ou générer tout type de couverture de test structurel et prouver la post-condition ou générer un contre-exemple. Une restriction sur le PSC global par quelques contraintes supplémentaires ou par une stratégie de recherche permet de limiter et d'orienter le PSC vers un ensemble spécifique de solutions pour répondre à des besoins variés de test.

Ensemble objectif de test (EOT)
En règle générale, l'objectif d'un test structurel consiste à analyser ou exécuter un ensemble de chemins qui répond à un critère de couverture simple ou combiné. Cet objectif peut être décomposé en un ensemble d'objectifs partiels exprimables en termes de noeuds de décision, noeuds de condition, ou noeuds d'instruction. Ainsi, un objectif partiel est une conjonction de conditions sur ces trois types de variables. Alors, nous pouvons représenter un objectif partiel par un ensemble de paires <$variable$,$valeur$> (i.e., l'objectif est l'exécution de Ns_2 exprimée par $\{<Ns_2,1>\}$). Un objectif partiel peut contenir des paires de la même classe ou des classes combinées. Dans notre approche, l'objectif de test est représenté par un EOT qui contient des objectifs partiels. Nous donnons les EOT pour les mesures de couverture les plus utilisées. Le recensement ci-après n'est pas exhaustif.

1. Couverture de toutes les instructions : chaque objectif partiel est un ensemble qui contient une seule paire composée d'une variable noeud d'instruction et la valeur 1. L'EOT contient tous les objectifs partiels possibles (i.e., $\{\{<Ns_i, 1>\}/\ 0 \leq i \leq 8\}$).
2. Couverture de toutes les décisions : chaque objectif partiel est un ensemble singleton qui contient une paire composée d'une variable noeud de décision et la valeur 1 ou -1. L'EOT contient tous les objectifs partiels possibles (i.e., $\{\{<Nd_i, 1>\}, \{<Nd_i, -1>\}/\ 0 \leq i \leq 5\}$).
3. Nous utilisons la même méthode pour générer l'EOT pour d'autres objectifs de test ou mesures de couverture : toutes conditions, conditions/decisions, multi-conditions/décisions. Nous utilison l'algorithme proposé dans [10] pour générer l'EOT de la couverture conditions/décisions modifiées.

Génération des données de test
Pour générer des données de test en fonction d'un objectif, nous devons d'abord construire l'EOT. Tant que cet ensemble n'est pas vide, on choisit un objectif partiel, on l'enlève et on fixe ses variables à leurs valeurs, puis nous cherchons une solution. Si le PSC n'a pas de solution, nous marquons cet objectif partiel comme irréalisable. En général, la preuve d'un PSC irréalisable est un problème indécidable, alors durant la recherche, nous montrons partiellement qu'il est irréalisable. Si le problème a une solution, nous vérifions si d'autres objectifs partiels sont satisfaits par cette solution, dans le cas échéant, nous supprimons ces objectifs partiels de l'EOT.

Analyse structurelle
Un jeu de données à analyser est représenté par un ensemble de données de test (EDT) qui contient des vecteurs de test $< val_1, ..., val_n >$, où val_i est la valeur du paramètre d'entrée numéro i. Pour mesurer ou analyser la couverture structurelle d'un jeu de données de test, nous devons d'abord construire l'EOT correspondant à l'objectif de test. Tant que l'EDT et l'EOT ne sont pas vides, on choisit un vecteur de données et on l'enlève de l'EDT, dans le PSC nous fixons les variables paramètres à leurs valeurs, puis nous cher-

TABLE 3 – Couverture toutes les décision : comparaison avec l'approche Gotlieb, et al. sur le programme *try_type*.

Decision		**Nd0**		**Nd1**		**Nd2**		**Nd3**		**Nd4**		**Nd5**	
Value		1	-1	1	-1	1	-1	1	-1	1	-1	1	-1
Time (s)	$CSP1$	0.01	0.01	0.77	0.01	0.01	0.01	0.01	>300	>300	>300	>300	>300
	$CSP2$	0.01	0.75	0.95	0.75	0.09	0.75	0.01	0.75	0.01	0.75	0.01	0.75

chons une solution. Si le problème n'a pas de solution, nous marquons ce vecteur hors domaine. Si le problème a une solution, nous vérifions s'il existe des objectifs partiels qui sont satisfaits par cette solution ; le cas échéant, nous supprimons ces objectifs partiels de l'EOT. Enfin, si l'EOT devient vide, alors une couverture de 100% est atteinte, sinon nous calculons le pourcentage de couverture. Pour le sous-ensemble de l'EOT qui n'est pas couvert, nous générons des données de test et les chemins qui ne sont pas couverts.

Prouver une post-condition

Nous formulons la contrainte équivalente de la post-condition à prouver en remplaçant les variables de PST par les variables équivalentes du PSC, puis nous insérons la négation de cette contrainte dans ce PSC [6]. Si le problème a une solution, alors il existe un chemin qui viole cette post-condition. La solution donne le chemin concerné et les données de test pour violer la post-condition. Si le problème n'a pas de solution, alors le post-condition est toujours satisfaite.

Génération des données de test pour une couverture k-chemins

Dans notre approche, un chemin d'exécution est une affectation des variables noeud de décision. Un premier chemin peut être couvert par la première solution atteinte par le solveur pour le CSP global. À partir de cette solution, nous pouvons construire le prédicat de ce chemin qui est une conjonction des expressions d'égalité entre les variables noeud de décision et leurs valeurs. L'insertion de la négation de ce prédicat dans le PSC global force le solveur à donner une autre solution correspondant à un chemin différent. Nous utilisons ce mécanisme : pour chaque solution trouvée, la négation de son prédicat est insérée dans PSC global, jusqu'à ce qu'il devienne irréalisable, ce qui signifie que tous les chemins faisables sont couverts.

Contrairement au principe basé sur l'EOT, chaque chemin (objectifs partiel) est un PSC à résoudre, ce qui signifie un PSC irréalisable pour chaque chemin infaisable, qui est également adoptée par plusieurs autres approches [6, 7, 18]. Notre approche n'a qu'un seul PSC irréalisable à résoudre.

Pour réaliser ce mécanisme, nous définissons une stratégie de recherche : dans un premier niveau de recherche l'énumération est faite sur les variables noeud de décision (Nd_i), dans un deuxième niveau de recherche l'énumération est faite sur les variables qui représentent les paramètres d'entrée. Une fois qu'une solution est trouvée, nous obligeons le solver à faire un retour-arrière vers le premier niveau.

5 Comparaisons avec l'état de l'art

Nos comparaisons sont faites en fonction du temps nécessaire pour générer un jeu de données de test pour un critère de couverture spécifique. Toutes les expériences ont été réalisées avec ILOG OPL Studio 3.7.1, sur un Windows XP, Intel Core 2 Duo 2 GHz, 2 Go de mémoire. Pour la première comparaison, nous avons limité le temps de recherche à 5 minutes.

Couverture toutes décisions

Pour comparer notre approche avec celle de Gotlieb et al. [12], nous avons traduit manuellement l'implémentation du programme triangle proposée par [15] qui est différente de celle utilisée par [12, 18] en deux PSC. Dans une première version, $PSC1$, nous avons utilisé l'approche proposée par Gotlieb et al. et dans une seconde version, $PSC2$, nous avons utilisé notre approche. La fonction *tri_type* contient six instructions de contrôle. Notre objectif est de satisfaire une couverture de toutes les décisions. Pour chaque PSC, nous avons créé un ensemble de douze contraintes équivalentes, une par branche, et avons ajouté à chaque version une contrainte à la fois. La Tab. 3 fournit les détails des résultats obtenus.

Pour les sept premières branches, le temps nécessaire pour résoudre le $PSC1$ est légèrement meilleur que le temps nécessaire pour résoudre notre $PSC2$, l'une des raisons étant que l'approche de Gotlieb et al. ne contient que des variables de PST, alors que notre approche contient également des variables de noeud, ce qui nécessite un délai supplémentaire pendant la recherche. Une autre raison est que ces sept branches sont faciles à couvrir (non-triangle, scalènes). Toutefois, notre approche est plus efficace dans les cinq dernières branches, qui sont plus compliquées à satisfaire (équilatéral, isocèle). Sur les douze branches, après cinq minutes d'attente pour chaque branche, $PSC1$ n'a pas pu générer des données de test pour couvrir les cinq branches qui représentent 40 % du PST, tan-

TABLE 4 – Couverture de tous les chemins : comparaison avec PathCrawler [18].

Program	k-chemins	#Ch. faisables	Notre Approche		PathCrawler [18]	
			#D. de test	temps (s)	#D. de test	Temps (s)
try_type	-	10	10	0.001	14	0.010
Merge	2	17	17	0.080	19	-
Merge	5	321	321	0.148	337	0.780
Merge	10	20481	20481	28.640	20993	116.000
Sample	3	240	240	0.060	241	0.270

TABLE 5 – Couverture de tous les chemins : comparaison avec PathCrawler [3] sur le programme *Merge*.

k-chemins	#Ch. faisables	Notre approche		PathCrawler [3]	
		#D. de test	Temps (s)	#D. de test	Temps (s)
2	17	17	0.080	19	0.330
5	321	321	0.148	337	0.800
10	12287	12287	18.163	12798	37.200
15	204931	204931	827.250	216371	876.350
All-Paths(19)	705431	705431	5486,953	705431	3407,980

dis que notre approche a généré des données de test pour chaque branche.

Couverture k-chemins

Nous avons utilisé notre approche de couverture de tous les k-chemins pour traduire manuellement les programmes présentes dans la Tab. 4, puis nous avons comparé nos résultats avec ceux rapportés dans [18] (Tab. 4) et [3] (Tab. 5).

Par définition, un vecteur de données de test qui est généré pour une couverture de k-chemins ne doit pas dépasser k itérations dans une boucle. Le mécanisme de génération de prédicat d'un chemin selon son identification partielle (chemin incomplet) utilisé par PathCrawler peut générer des données de test qui dépassent cette limite. Ces données de test superflues expliquent la différence du nombre de données de test par rapport à notre approche. Nous avons généré le nombre exact de données de test équivalent au nombre de chemins réalisables, alors que PathCrawler a généré des données de test supplémentaires (512 données de test superflues pour $k = 10$), mais ces données ne sont pas compatibles avec la valeur de k choisie.

En termes de temps d'exécution, notre approche a prouvé son efficacité en particulier pour les deux derniers programmes (*Merge* et *Sample*) où il est presque cinq fois plus rapide que PathCrawler, ce qui veut dire que même si nous supposons que notre machine est 5 fois plus rapide que celle utilisée pour les expérimentations de PathCrawler, ce qui loin d'être le cas, nos résultats restent comparables. Vu que nous avons traduit manuellement les programmes, nous avons ignoré le temps nécessaire pour cette tâche parce qu'il n'est pas significatif par rapport au temps nécessaire pour la résolution du PSC global.

La génération des données de test pour couvrir tous les chemins sans limiter le nombre d'itérations présentée dans Tab. 5 montre que PathCrawler est légèrement plus efficace. Pour une valeur de k supérieure à dix, le nombre d'itérations des deux dernières boucles du programme *Merge* ne dépasse pas dix. Mais notre approche utilise une seule valeur pour k, ce qui signifie un nombre supplémentaire de chemins infaisables à prouver. Pour un k inférieur ou égal à dix, notre approche est nettement meilleure que PathCrawler [3]. En termes pratiques, pour un PST qui contient des boucles, le test structurel limite le nombre d'itérations à un petit nombre qui est généralement égal à deux ou trois. Ainsi un critère de couverture tous-chemins sans limitation sur le nombre d'itérations n'est pas réaliste [16]. Nous observons également que, pour $k = 15$, PathCrawler a généré un ensemble de 11.440 cas de test supplémentaires qui doivent alors être appliqués inutilement.

6 Conclusion

Dans cet article, nous avons présenté une nouvelle approche pour différentes applications de test structurel. Il s'agit d'une approche qui combine les applications de test structurel les plus utilisées. Sa nouveauté réside dans l'utilisation de la PC pour l'analyse structurelle, la combinaison d'un nombre important d'applications de test structurel, la combinaison de tous les critères de couverture structurelle qui sont basés sur le flot de contrôle et la modélisation par contraintes

d'un GFC. Une nouvelle restructuration du GFC et une classification de ses noeuds ont été données. Nous avons montré que la modélisation d'un PST combiné à son GFC par un PSC conserve sa sémantique structurelle et peut donc répondre aux différents besoins de test structurel : analyse structurelle, génération des données de test et preuve de post-condition. Nous avons déjà donné, à la section 4.4, certaines façons dont notre approche peut être utilisée pour répondre aux besoins variés de test structurel. Notre approche peut être utilisée pour automatiser le processus de test structurel, réduire la taille d'un ensemble de données de test et réduire ainsi le temps nécessaire pour tester un système critique.

Les résultats obtenus en comparant notre approche à des approches de la littérature sur différents benchmarks sont très prometteurs, ce qui peut être considéré comme un très bon point de départ pour une automatisation complète du processus de test structurel sur la base de PC. L'efficacité de notre approche est fortement dépendante des performances du solveur, ces dernières sont limitées par le nombre de variables et la taille de leurs domaines. Dans le futur, nous allons nous concentrer à la fois sur l'implantation d'un outil complet qui mettra en oeuvre cette approche et sur son extension pour dépasser ses limites actuelles. En particulier, nous aimerions traiter les pointeurs, les nombres à virgule flottante et les appels aux fonctions, qui sont une extension difficile mais importante pour traiter des systèmes industriels.

Références

[1] Douglas N. Arnold. The explosion of the ariane 5. `http://www.ima.umn.edu/~arnold/disasters/ariane.html`, 2000. [Online ; accessed 09-Mai-2011].

[2] S. Bardin and P. Herrmann. Structural testing of executables. In *International Conference on Software Testing, Verification and Validation*, pages 22–31, 2008.

[3] B. Botella, M. Delahaye, N. Kosmatov, M. Roger P. Mouy, and N. Williams. Automating structural testing of c programs : Experience with pathcrawler. In *Fourth International Workshop on the Automation of Software Test*, pages 70–78.

[4] B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et sciences informatiques*, 21 :21–45, 2002.

[5] M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 14 :1684–1698, 1994.

[6] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920, pages 0302–9743, 2006.

[7] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. 15 :238–264, 2010.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13 :451–490, 1991.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9 :319–349, 1987.

[10] K. A. Foster. Sensitive test data for logic expressions. *ACM singsoft software engineering notes*, 9 :120–125, 1984.

[11] A. Gotlieb. Euclide : A constraint-based testing framework for critical c programs. In *International Conference on Software Testing, Validation and Verification*, pages 151–160, 2009.

[12] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. *Computational Logic*, 1861 :399–413, 2000.

[13] J. Jaffar and M. J. Maher. Constraint logic programming - a survey. *Logic Programming*, 20 :503–581, 1994.

[14] J. Kong and H. Yan. Comparisons and analyses between rtca do-178b and gjb5000a, and integration of software process control. In *International Conference on Advanced Computer Theory and Engineering*, volume 6, pages 367–372, 2010.

[15] P. McMinn. Search-based software test data generation : a survey. *Software Testing Verification & Reliability*, 14 :105–156, 2004.

[16] G. J. Myers. *The art of software testing.* John Wiley and Sons, 1979.

[17] RTCA. Software consideration in airborne systems and equipment certification. *Software verification process*, 1992.

[18] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*, volume 3463, pages 281–292, 2005.