Utilisation des contraintes pour la génération automatique de cas de test structurels

Bernard Botella[*] — Arnaud Gotlieb[*] — Claude Michel[**] — Michel Rueher[**] — Patrick Taillibert[*]

[*] *Thales Systèmes Aéroportés, Centre Charles Nungesser*
2, av. Gay-Lussac
78851 Elancourt Cedex
{Bernard.Botella, Arnaud.Gotlieb, Patrick.Taillibert}@fr.thalesgroup.com
[**] *Université de Nice–Sophia-Antipolis, I3S/CNRS,*
ESSI, 930, route des Colles - B.P. 145
06903 Sophia-Antipolis
{cpjm, rueher}@essi.fr — http://www.essi.fr/~ rueher

RÉSUMÉ. *Une des difficultés majeures pour l'automatisation du processus de test structurel réside dans la production automatique des cas de test, c'est-à-dire la détermination d'un ensemble de valeurs d'entrée pour lesquelles un point choisi du programme sera exécuté. Nous présentons ici une nouvelle méthode où ce problème est transformé en un problème de résolution de contraintes. La traduction du programme initial en un système de contraintes est effectuée en utilisant la forme SSA et les dépendances de contrôle. Des opérateurs spécifiques, implantés avec des contraintes gardées, ont été introduits pour autoriser le traitement de contraintes propres à cette application.* INKA*, le système prototype qui a été développé, permet de traiter des programmes utilisant un sous-ensemble significatif des constructions du langage* `C` *(e.g., avec des tableaux, instructions itératives "while", certains pointeurs). Les premiers résultats expérimentaux sur des exemples académiques montrent que* INKA *est concurrentiel avec les méthodes traditionnelles.*

ABSTRACT. *The generation of test data is one of the main difficulties of the unit testing process of software in industrial applications. Thus, a major challenge consists in generating test data automatically, i.e., in finding input values for which a selected point in a procedure is executed. We introduce here an original framework where the latter problem is transformed into a constraint solving problem. The initial program is translated into a constraint system by means of the well known "single static assignment" form and control dependencies. Specific operators, based upon entailment techniques, have been introduced to tackle this kind of applications.* INKA*, the prototype system we have developed, allows us to handle programs that use a significant subset of the features of the* `C` *language (e.g., arrays, "while" instructions, some pointers). First experimental results show that* INKA *is competitive with traditional methods.*

MOTS-CLÉS : *test structurel, cas de test, programmation par contraintes, contraintes gardées*

KEYWORDS: *structural testing, test data, constraint programming, entailment techniques*

1. Introduction

La preuve et le test sont les deux principales méthodes qui peuvent être utilisées pour s'assurer de la correction d'un logiciel. La preuve exige une spécification formelle qui est difficile à élaborer et de ce fait rarement disponible. De plus, les limites des méthodes de démonstration automatique restreignent sensiblement le champ d'application des techniques de preuve de programme. Celles-ci ne sont donc souvent utilisées que sur des parties de logiciel et en complément des méthodes de test. Le test reste donc une des étapes essentielles dans le cycle de vie d'un logiciel.

Parmi les différentes méthodes de test, on peut distinguer deux grandes familles : le *test fonctionnel* qui s'appuie sur la spécification des programmes, et le *test structurel* qui est basé sur l'analyse du code source. La complémentarité de ces deux approches est aujourd'hui bien admise [ZHU 97]. Le test structurel est tout particulièrement requis pour les logiciels critiques qui doivent satisfaire certains critères imposés par les organismes normatifs (ISO, Afnor, DoD,...). Il représente un coût non négligeable dans le développement des logiciels lorsque les exigences de sécurité sont importantes.

Le test structurel peut se décomposer en deux étapes :

– L'identification d'instructions dont l'exécution est nécessaire pour satisfaire un critère de couverture (e.g., tous les chemins, toutes les branches, toutes les instructions) ;

– La génération d'un cas de test (i.e., un ensemble de valeurs pour les variables d'entrée) qui garantit que l'instruction sélectionnée sera effectivement exécutée.

Dans cet article nous nous intéressons au second problème. Plus précisément, nous cherchons à générer automatiquement un cas de test exécutable pour une instruction donnée. Ce problème, appelé ATDG (Automatic Test Data Generation), est un problème difficile : il est indécidable dans le cas général car il se réduit au problème de la "halte" [WEY 79]. Nous présentons dans cet article une nouvelle démarche [1] où ce problème est transformé en un problème de résolution de contraintes. La formulation sous forme d'un système de contraintes autorise la recherche d'un cas de test sans choisir un chemin au préalable. Il s'agit là d'un point clé car il existe de très nombreux chemins non exécutables dans un programme. Mais, avant de détailler les caractéristiques de notre méthode, nous allons rapidement rappeler les limites des méthodes existantes.

1.1. *Limites des approches classiques*

Les méthodes classiques peuvent se classer en trois catégories :

– *La génération aléatoire* [NTA 98] qui essaie des valeurs de manière aveugle jusqu'à ce que le point sélectionné soit atteint ;

1. Cet article est une version étendue des travaux que nous avons présentés dans [GOT 98, GOT 00b].

– *L'exécution symbolique* [KIN 76, DEM 93] qui remplace les paramètres d'entrée d'une procédure par des valeurs symboliques et évalue symboliquement l'ensemble des instructions sur un chemin du graphe de contrôle qui passe par l'instruction sélectionnée ;

– *Les méthodes "dynamiques"* [KOR 90, FER 96, GUP 99, GUP 00] qui partent d'un cas de test initial et le "corrige" (en utilisant des techniques d'optimisation) jusqu'à atteindre le point sélectionné.

L'avantage du test aléatoire est qu'il ne nécessite aucune analyse du programme : il suffit de vérifier que le point sélectionné est effectivement atteint. Cette méthode — qui conduit à générer un nombre très important de cas de test— est bien adaptée pour les programmes dont le temps d'exécution est négligeable et pour lesquels il existe une spécification formelle autorisant une vérification automatique de l'oracle (e.g., [BOU 99]). Sa limite majeure réside dans son incapacité à exercer des parties très peu probables du programme.

Les méthodes symboliques ne savent pas traiter correctement les tableaux et les pointeurs. Un problème majeur de l'exécution symbolique réside aussi dans la simplification et la résolution de l'expression algébrique qui est générée. Ce problème est particulièrement délicat si le chemin sélectionné est non exécutable. Or, dans un programme il existe de très nombreux chemins non exécutables ; si le programme contient des instructions itératives, le nombre de chemins non exécutables peut même être infini. Déterminer statiquement si un chemin est non exécutable est un problème difficile qui est indécidable dans le cas général [AHO 86, WEY 79].

La méthode dynamique a été proposée pour remédier aux limites de l'approche symbolique. Un premier cas de test est en général généré de manière aléatoire. Si le chemin associé à ce cas de test ne passe pas par le point sélectionné, d'autres chemins sont dérivés à partir de ce premier chemin et des informations issues du graphe du flot de contrôle et du graphe de flot de données. Cette approche explore ainsi un ensemble de chemins jusqu'à ce que le point sélectionné soit atteint. En général, les méthodes associées à cette approche n'arrivent pas à identifier des parties de code non exécutable. De plus, elles sont fortement dépendantes des techniques d'optimisation et des heuristiques utilisées pour changer de chemin. Il en est de même des méthodes qui reposent sur les algorithmes génétiques [PAR 99].

1.2. *Apports des techniques de programmation par contraintes*

Un système de programmation par contraintes est défini par un ensemble de variables, un ensemble de domaines associés à ces valeurs (i.e., les valeurs admissibles) et un ensemble de relations entre ces variables. Un point clé de la programmation par contraintes réside dans le fait que *la résolution des problèmes est dirigée par les données* : les valeurs pour lesquelles il est possible, en exploitant les propriétés du domaine de calcul, de détecter qu'elles ne peuvent pas appartenir à une solution sont

progressivement éliminées des domaines, et la recherche s'effectue à partir des valeurs les plus prometteuses.

Les contraintes offrent un cadre qui permet une généralisation des méthodes dynamiques. En effet, les informations issues du graphe de données sont des relations dont la traduction en contraintes est immédiate. Les contraintes conditionnelles [HEN 98] permettent quant à elles d'exprimer les informations issues du graphe du flot de contrôle. Les informations concernant les points à atteindre (ou un chemin défini par une séquence de points à atteindre) conduisent à rendre déterministes certaines contraintes conditionnelles générées à partir du graphe du flot de contrôle. La résolution du système de contraintes restreint progressivement le domaine des variables à des valeurs qui correspondent à un cas de test permettant d'atteindre les points sélectionnés. Il est important de noter qu'aucun chemin n'est défini a priori : c'est la réduction du domaine des variables et l'élimination des contraintes conditionnelles [2] qui permettent progressivement de déterminer un ensemble de chemins valides.
La rapidité de convergence vers la solution dépend toutefois de la stratégie d'élimination des contraintes conditionnelles.

Les techniques de programmation logique avec contraintes ont été utilisées pour la génération de cas de test fonctionnels. Dans ce cadre, les contraintes ont permis un traitement des valeurs limites plus réalistes et la détection précoce des sous-problèmes ne possédant pas de solution dans le domaine des valeurs autorisées [ARN 99, MEU 98]. Le système GATEL [MAR 00] qui permet la génération de séquences de test à partir de description en LUSTRE [3] repose aussi sur la programmation logique avec contraintes : les constructions du langage LUSTRE sont traduites en contraintes sur des variables booléennes ou à valeurs dans des intervalles d'entiers ; la génération des séquences consiste à résoudre ces contraintes à l'aide de techniques de dépliage et de programmation logique avec contraintes (symboliques). Le système CASTING [AER 98, LIO 97] propose une méthode semi-automatique qui pourrait être étendue à la génération de tests structurels. AUTOFOCUS [PRE 01, LöT 00b, LöT 00a] est un système basé sur l'exécution symbolique et la programmation par contraintes qui requiert une modélisation abstraite du programme.

Nous présentons dans cet article une méthode, basée sur la programmation par contraintes, pour la génération de cas de test structurels pour des programmes écrits dans un langage impératif. La sous-section suivante illustre ses apports sur un petit exemple.

---

2. Schématiquement, une contrainte conditionnelle est éliminée dès qu'il est possible de prouver ou de réfuter sa prémisse.
3. LUSTRE [HAL 91] est un langage formel pour les applications à flots de données synchrones. Une description LUSTRE peut être vue soit comme une spécification formelle exécutable, soit comme le code source du programme sous test.

```
unsigned int foo(unsigned int x, unsigned int y)
    unsigned int z,t,u;
1.  { z = x * y;
2.    t = 2 * x;
3.    if (x < 4)
4.      u = 10;
      else
5.      u = 2;
6.    if (z ≤ 8)
7.      {  if (u ≤ x)
8.           {  t = t − y;
9.              if (t ≤ 20)
10.                { ...
```

Figure 1. *Le programme foo*

1.3. *Un exemple illustratif*

Pour illustrer l'apport de notre approche, nous allons utiliser le programme jouet de la figure 1. L'objectif est de générer un cas de test, i.e., un couple de valeurs pour (x,y) pour lequel l'instruction 10 sera exécutée.

La première étape consiste à générer un système de contraintes à partir du programme initial. Cette transformation s'effectue à partir d'une analyse des dépendances de contrôle associées au programme (cf. figure 2) et à l'aide d'une technique bien connue en compilation, la forme "Static Single Assignment" (SSA) [CYT 91]. Schématiquement, la forme SSA (cf. section 3.1) consiste à introduire des nouvelles variables pour éviter qu'une même variable ne soit modifiée à différents endroits du programme ; des ϕ-fonctions expriment les dépendances entres les différentes variables dues au contrôle. Pour atteindre le point 10 du programme `foo`, nous allons ainsi générer le système de contraintes suivant :

$$\sigma = (x,y,z,t_1,t_2,u \in (0..2^{32}-1) \wedge \\ (z = x * y) \wedge (t_1 = 2 * x) \wedge (z \leq 8) \wedge (u \leq x) \wedge (t_2 = t_1 - y) \wedge (t_2 \leq 20)$$

Les variables t_1 et t_2 correspondent à différents renommages de la variable t introduits par la forme SSA. Une valeur entre 0 et $2^{32}-1$ est associée aux variables de type `unsigned int`.

La résolution par contraintes repose sur une mise en œuvre alternée de :

– Techniques de filtrages, i.e., réduction du domaine des variables en ne considérant qu'une seule contrainte à la fois ;

– Déduction de nouvelles contraintes ;

– Enumération de valeurs pour certaines variables en utilisant différentes heuristiques.

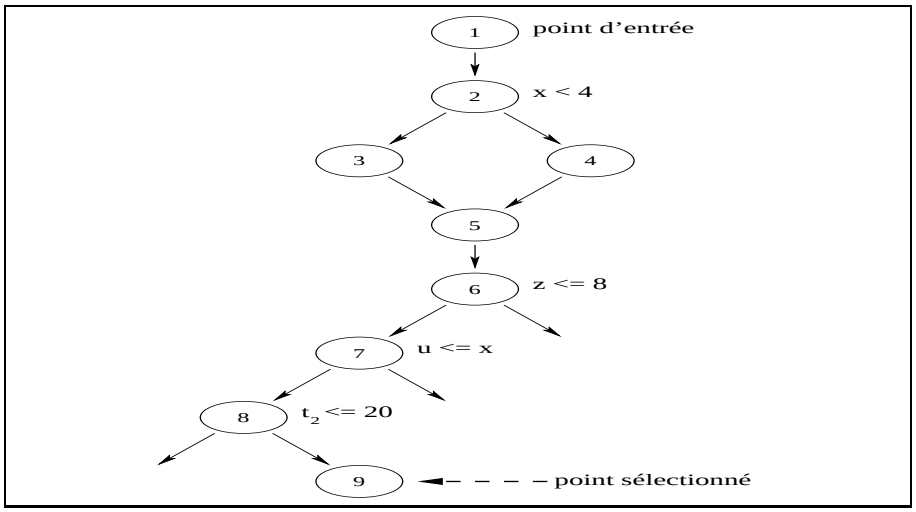

Figure 2. *Le graphe du flot de contrôle du programme* `foo`

Sur notre exemple, le filtrage par consistance d'intervalles (cf. définition 2.2) ne permet pas de réduire de manière significative les domaines des variables : le domaine de x sera réduit à $0..2^{16}-1$ et celui de y reste inchangé. Le domaine de recherche associé à (x,y) contient donc $2^{16} \times 2^{32}$ valeurs.
Toutefois, il est possible de déduire d'autres informations du programme. Par exemple, à partir de la première instruction "*if_then_else*" (lignes 3,4,5) il est possible de déduire les deux contraintes suivantes :

$(x \geq 4 \wedge u = 2)$ est vrai si $\neg(x < 4 \wedge u = 10)$ est vrai
$(x < 4 \wedge u = 10)$ est vrai si $\neg(x \geq 4 \wedge u = 2)$ est vrai

Ces "contraintes conditionnelles" sont ajoutées automatiquement lors de l'analyse du flot de contrôle. Les contraintes conditionnelles sont réécrites en contraintes non conditionnelles lorsque leur prémisse est impliquée par les autres contraintes non conditionnelles. Ainsi, comme la relation $\neg(x < 4 \wedge u = 10)$ est impliquée par la contrainte $u \leq x$ de σ, la relation $(x \geq 4 \wedge u = 2)$ sera ajoutée à notre ensemble de contraintes. Le filtrage par consistance d'intervalles de $(x \geq 4 \wedge u = 2) \wedge \sigma$ permet maintenant de réduire le domaine de x à [4..11] et celui de y à [0..2]. L'énumération sur la première valeur de y produit finalement le cas de test $(x = 4, y = 0)$ sans qu'il ait été nécessaire de sélectionner explicitement un chemin dans le graphe du flot de contrôle.

Il est important de noter qu'un sur-ensemble des chemins exécutables qui permettent d'atteindre le point sélectionné est caractérisé par les contraintes non conditionnelles. Cet ensemble de chemins est progressivement réduit lors de la réécriture

des contraintes conditionnelles [4] et par la réduction des domaines (ces derniers sont utilisés pour déterminer si les prémisses d'une contrainte conditionnelle sont vérifiées). Un système de contraintes inconsistant correspond à un chemin non exécutable.Cette inconsistance n'est évidemment pas toujours détectable : dans certains cas, les temps de calcul peuvent devenir prohibitifs et si le programme contient des instructions itératives, le système de contraintes peut devenir infini.

1.4. *Synthèse de notre contribution*

Nous proposons dans cet article une démarche originale pour générer des cas de test logiciels, i.e., des données d'entrée pour lesquelles un point sélectionné dans une procédure sera exécutée. Schématiquement, elle consiste à transformer statiquement une procédure en un système de contraintes en utilisant les techniques statiques d'assignation unique et les dépendances de contrôle du programme. La résolution du système de contraintes obtenu permet de vérifier s'il existe au moins un chemin exécutable passant par le point choisi et permet de produire des cas qui correspondent à un ou plusieurs de ces chemins. Elle permet aussi de détecter très tôt certaines instructions non atteignables. Un système prototype a été développé sur un sous-ensemble restreint du langage C et a permis une première validation de notre démarche.

1.5. *Plan de l'article*

La suite de cet article est organisée de la manière suivante. Les notations et les définitions de base sont introduites dans la section 2. La mise sous forme SSA d'un programme écrit dans un langage impératif comme C et la génération du système de contraintes sont détaillées dans la section 3. La section 4 explique le mode de traitement des contraintes conditionnelles et plus généralement fournit les grandes lignes de la résolution du système de contraintes. Quelques extensions du système INKA (e.g., traitement des pointeurs, prise en compte des flottants) sont décrites dans la section 5. Enfin, la section 6 présente quelques résultats expérimentaux.

2. Notations et définitions

Dans le schéma CLP(FD) [JAF 86, JAF 94], un *domaine* est un ensemble fini non vide d'entiers. Une variable associée à un domaine est une *FD_variable* ; elle est représentée par une majuscule. Une *contrainte primitive* est construite à l'aide des variables, des domaines, de l'opérateur $\in$, des opérateurs arithmétiques $\{+, -, \times$,*div, mod*$\}$ [5] et des relations $\{>$, $\geq$, $=$, $\neq$, $\leq$, $<\}$. Par la suite, c, éventuellement indicé, désignera exclusivement une contrainte primitive. La négation d'une contrainte

4. La réécriture des contraintes conditionnelles en contraintes non conditionnelles correspond, dans le pire des cas, à une forme d'énumération des chemins

5. `div` et `mod` représentent le quotient et le reste dans la division Euclidienne.

primitive, notée $\neg c$ est également une contrainte primitive. σ est une conjonction de contraintes primitives et non primitives. Parmi ces dernières, nous utiliserons plus particulièrement la contrainte *élément/3* pour le traitement des tableaux et les *contraintes gardées* [CAR 94] pour le traitement des instructions conditionnelles et du "*while*".

La contrainte `element(X, LISTE, Y)` contraint le $X^{ième}$ élément de la liste LISTE à être égal à la valeur Y.
Une contrainte conditionnelle, aussi appelée contrainte gardée, est notée $C_1 \longrightarrow C_2$, où C_1 (la garde) et C_2 sont des contraintes, et dont la sémantique opérationnelle est définie par les règles suivantes :

– Si C_1 est impliquée par les contraintes non conditionnelles de σ, la contrainte $C_1 \longrightarrow C_2$ est retirée de σ et C_2 est ajoutée à σ

– Si $\neg C_1$ est impliquée par les contraintes non conditionnelles de σ, la contrainte $C_1 \longrightarrow C_2$ est retirée de σ ;

– Si ni C_1 ni $\neg C_1$ ne sont impliquées par σ, la contrainte $C_1 \longrightarrow C_2$ est suspendue, c'est-à-dire qu'elle sera reconsidérée lorsque plus d'informations sur C_1 ou $\neg C_1$ seront disponibles.

Bien que décidable, la preuve de l'implication est NP-complète. C'est pourquoi deux méthodes partielles ont été définis [HEN 98] : *l'ac–implication* et *l'intervalle–implication* qui s'appuient respectivement sur la *consistance d'arc* et la *consistance d'intervalle* que nous définissons ci-dessous.

Les filtrages basés sur les consistances partielles sont un des points clé des techniques de programmation par contraintes. Informellement, il s'agit de méthodes qui permettent de détecter qu'il est impossible qu'une variable prenne une valeur donnée en ne considérant qu'une contrainte. Par exemple, si X_1 et X_2 ont respectivement pour domaines $D_1 = \{1,2,3\}$ et $D_2 = \{1,4,5\}$, alors la contrainte $X_1 < X_2$ permet d'éliminer la valeur 1 de D_2. Nous utiliserons deux consistances locales : la *consistance d'arc* et la *consistance d'intervalle*.

Soient $X_1, \ldots, X_n$ un ensemble de FD_variables, $D_1, \ldots, D_n$ leur domaine et C une contrainte définie sur $X_1, \ldots, X_n$.

Définition 2.1 *(consistance d'arc)*
Une contrainte C est arc-consistante [MAC 77] si pour chaque variable X_i et pour chaque valeur $v_i \in D_i$ il existe des valeurs $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ dans $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ telles que $C(v_1, \ldots, v_n)$ soit vérifiée.
Un ensemble de contraintes σ est arc–consistant si pour chaque contrainte C dans σ, C est arc consistante.

La consistance d'intervalle utilise D*, une approximation d'un domaine fini D, qui correspond à l'ensemble des entiers compris entre la borne inférieure et la borne supérieure de D : D*={x∈N, min(D) ≤x≤max(D)}. Plus formellement :

Définition 2.2 *(consistance d'intervalle)*
Une contrainte C est intervalle-consistante si pour chaque variable X_i et chaque

valeur $v_i \in \{min(D_i), max(D_i)\}$ il existe des valeurs $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ dans les domaines $D_1^*, \ldots, D_{i-1}^*, D_{i+1}^*, \ldots, D_n^*$ tel que $C(v_1, \ldots, v_n)$ soit vérifiée.
Un ensemble de contraintes σ est intervalle-consistant si pour chaque contrainte C dans σ, C est intervalle-consistant.

Dans [HEN 98], deux méthodes partielles ont été proposées pour détecter une implication :

Définition 2.3 *(ac–implication)*
Une contrainte $C(X_1, \ldots, X_n)$ peut être déduite d'un ensemble de contraintes σ par ac–implication pour $D_1, \ldots, D_n$ ssi pour tout $v_1, \ldots, v_n$ dans $D_1, \ldots, D_n$, on a $C(v_1, \ldots, v_n)$.

Définition 2.4 *(intervalle–implication)*
Une contrainte $C(X_1, \ldots, X_n)$ peut être déduite de σ par intervalle–implication pour $D_1, \ldots, D_n$ ssi pour tout $v_1, \ldots, v_n$ dans $D_1^*, \ldots, D_n^*$, on a $C(v_1, \ldots, v_n)$.

Nous utiliserons également *l'abs-implication*, une autre méthode partielle pour détecter l'implication s'appuyant sur une preuve par réfutation. Cette relaxation de l'implication qui nécessite le calcul de la négation de la contrainte et la restauration de l'ensemble de contraintes σ après l'étape de filtrage, s'est avérée plus efficace pour notre application que les deux autres relaxations de l'implication mentionnées ci-dessus.

Définition 2.5 *(abs-implication)*
Une contrainte C est abs-impliquée par σ ssi le filtrage de $\sigma \wedge \neg C$ par la consistance d'arc ou par la consistance d'intervalle génère un domaine vide.

On peut remarquer que ces trois propriétés sont plus fortes que l'implication et donc plus faciles à prouver que l'implication.

3. Génération du système de contraintes

Soit P une procédure d'un langage impératif quelconque et soit n un point de P [6]. La résolution du problème de la génération automatique de cas de tests nécessite la détermination d'un vecteur de valeurs d'entrée de P (paramètres ou variables globales) telles que l'exécution de P pour ce vecteur engendre le passage par le point n.

Nous allons décrire la procédure que nous proposons dans le cadre d'un langage impératif comprenant des données entières simples, des tableaux, et des instructions de contrôle "*if*" et "*while*". Nous aborderons dans la section 5 les problèmes posés par le traitement des flottants et des pointeurs.

6. Un point correspond à une instruction d'un bloc ou une décision.

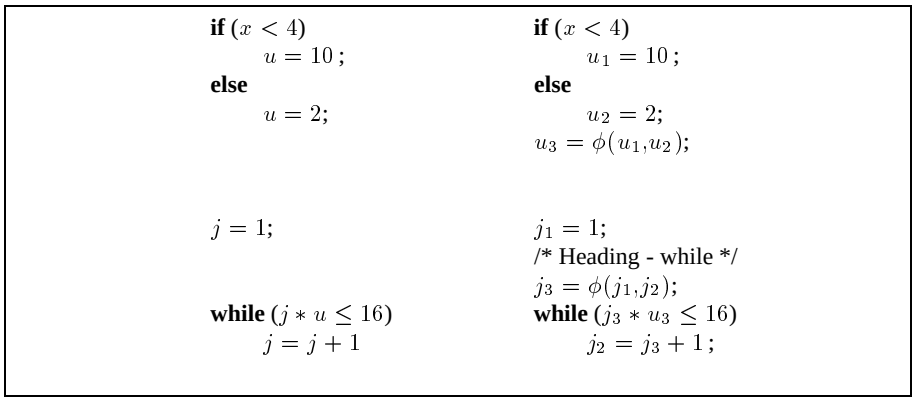

Figure 3. *Forme SSA des instructions de contrôle*

La génération du système de contraintes s'effectue en trois étapes :

1. Transformation du programme en une forme statique à assignation unique (forme SSA)[CYT 91] ;

2. Transformation de la forme SSA en un ensemble de contraintes CLP(FD) ;

3. Génération des contraintes associées au point sélectionné.

3.1. *Forme statique à assignation unique*

La forme statique à assignation unique d'un programme impératif est une version équivalente de ce programme dans laquelle chaque variable n'est définie qu'une fois et n'est donc jamais modifiée. La forme SSA d'un bloc d'instruction simple (sans branchement) s'obtient par simple renommage des variables ($i = i + 1$ devient $i_2 = i_1 + 1$). Les structures de contrôle nécessitent l'introduction d'une affectation particulière appelée ϕ-fonction, destinée à combiner différentes définitions de variables. Une ϕ-fonction est placée à un point de convergence du flot de contrôle et permet de savoir quel chemin a été effectivement suivi. Ainsi, dans le cas de l'instruction `if_then_else` (voir figure 3), la ϕ-fonction $\phi(u_1,u_2)$ retourne u_1 si le flot de contrôle a traversé la branche *then*, u_2 sinon. Dans le cas d'une instruction itérative comme le `while` la ϕ-fonction $\phi(j_1,j_2)$ est ajoutée dans un entête spécial et évaluée à chaque pas de l'itération. On notera que l'assignation unique ainsi obtenue est purement statique (dans le code source du programme, chaque variable n'est définie qu'une fois) ; lors de l'exécution, les variables peuvent être modifiées plusieurs fois (j_3 sur l'exemple de la figure 3) ce qui nécessitera des précautions particulières lors de la génération des contraintes.

3.2. *Génération du programme CLP*

L'idée essentielle est de transformer chaque instruction de la forme SSA du programme pour lequel on souhaite générer les cas de test en une contrainte pour former un programme CLP. Il sera alors possible d'exploiter la propriété de réversibilité des contraintes pour remonter aux valeurs des paramètres recherchés. Une clause est ainsi engendrée pour chaque procédure P du programme. La tête de clause possède 5 arguments :

– une liste de FD_variables correspondant aux paramètres de P,

– une liste de FD_variables correspondant aux variables globales référencées par P,

– une liste de FD_variables correspondant aux variables locales utilisées dans les décisions de P (qui sera utilisée pour définir le point du graphe de contrôle pour lequel le test est généré),

– une liste de FD_variables correspondant aux variables globales définies dans P ,

– une FD_variable correspondant à l'expression retournée par P.

Le processus de transformation est le suivant :

3.2.1. *Déclarations*

La déclaration de type de la variable x est traduite en une contrainte primitive de la forme : $X_i \in Min_T..Max_T$ où Min_T (resp. Max_T) est la valeur minimum (resp. maximum) du type T.

3.2.2. *Tableaux*

La forme SSA définit deux fonctions spécifiques pour la prise en compte des tableaux : *access*(a_0,k) dont la valeur retournée est celle du $k^{\text{ième}}$ élément du tableau a_0, et *update*(a_0,j,w) qui retourne un tableau identique à a_0, excepté pour l'élément d'indice j dont la valeur est w. Ces deux fonctions sont transformées à l'aide de la contrainte `element` décrite au chapitre précédent :

1. une utilisation de valeur de tableau $v =$ *access*(a_0,k) est transformée en : `element`(K,A_0,V) ;
2. une définition $a_1 =$ *update*(a_0,j,w) est transformée en : `element`(J,A_1,W) $\wedge$ $\bigwedge_{I \neq J}$[`element`(I,A_0,V) $\wedge$ `element`(I,A_1,V)].

Ces contraintes ne sont effectives que lorsque la taille du tableau est connue.

3.2.3. *Instruction "if_then_else"*

L'instruction "*if_then_else*" est transformée en utilisant une contrainte non primitive, nommée "*ite*", qui exprime la disjonction exclusive qu'engendre cette instruction : `ite`(c, $C_1 \wedge \ldots \wedge C_n$, $C'_1 \wedge \ldots \wedge C'_m$) est vraie ssi $(c \wedge C_1 \wedge \ldots \wedge C_n)$ ou $(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$ est vraie, où c est une contrainte primitive, et $C_1, \ldots, C_n, C'_1, \ldots, C'_m$

des contraintes primitives ou non. Ainsi, l'instruction de la figure figure 3 est transformée en : `ite`($X < 4$, $U_1 = 10 \wedge U_3 = U_1$, $U_2 = 2 \wedge U_3 = U_2$).

La sémantique opérationnelle de la contrainte non primitive "*ite*" est définie par les règles suivantes :

Proposition 3.1 `ite/3` *(sémantique opérationnelle)*
`ite`*(*c, $C_1 \wedge \ldots \wedge C_n$, $C'_1 \wedge \ldots \wedge C'_m$*) est réduite aux quatre contraintes gardées suivantes :*

1. $c \longrightarrow C_1 \wedge \ldots \wedge C_n$
2. $\neg c \longrightarrow C'_1 \wedge \ldots \wedge C'_m$
3. $\neg(c \wedge C_1 \wedge \ldots \wedge C_n) \longrightarrow (\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$
4. $\neg(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m) \longrightarrow (c \wedge C_1 \wedge \ldots \wedge C_n)$

Les deux premières contraintes gardées sont la traduction directe de la sémantique de l'instruction "*if_then_else*". Les deux dernières sont basées sur l'idée de réfutation. Elles ont pour but de permettre un élagage plus efficace de l'arbre de recherche en prenant en compte le fait que si l'une des deux branches est inconsistante, l'autre est obligatoirement vérifiée (c et $\neg c$ ne sont rajoutées aux gardes que pour faciliter la détection d'inconsistance par la technique de l'abs-implication).

3.2.4. *Instruction "while"*

La forme SSA d'une instruction "*while*" est la suivante :
$\vec{v_2} = \phi(\vec{v_0},\vec{v_1})$ *while* (c) $\{C_1; \ldots ; C_p\}$ mais cette notation est un peu trompeuse. Il faut en effet comprendre que la fonction ϕ est évaluée à chaque tour de boucle ; $\vec{v_0}$ est le vecteur des variables d'entrée de l'instruction "*while*" lors de l'entrée dans la boucle et $\vec{v_1}$ le vecteur des variables définies dans le corps de la boucle après chaque tour. La fonction retourne $\vec{v_2}$, le vecteur des variables utilisées à la fois à l'intérieur et à l'extérieur de la boucle.

La forme SSA associée à une instruction "*while*" ne permet pas une traduction immédiate de cette instruction en contraintes. En effet, l'assignation unique est statique (une même variable n'apparaît qu'une fois en partie gauche d'une affectation), mais à l'exécution, une affectation destructrice a lieu à chaque tour de boucle, ce qui ne peut évidemment pas être le cas d'une contrainte. Pour résoudre ce problème, nous générons non pas simplement une contrainte mais plutôt un programme qui génère des nouvelles variables et contraintes à chaque itération.

Nous avons ainsi défini l'opérateur *w*/5 dont la sémantique opérationnelle est définie par les règles suivantes :

Proposition 3.2 `w/5` *(sémantique opérationnelle)*
`w`*(*c,$\vec{V_0}$,$\vec{V_1}$,$\vec{V_2}$,$C_1 \wedge \ldots \wedge C_p$*) est réduit aux quatre contraintes gardées suivantes* [7] *:*

1. $c \longrightarrow (C_1 \wedge \ldots C_p \wedge w(c,\vec{V_1},\vec{V_3},\vec{V_2},C_1 \wedge \ldots \wedge C_p))$

7. Par souci de simplicité, nous ne distinguerons pas ici c des contraintes générées à partir de c en renommant certaines variables

2. $\neg c \longrightarrow \vec{V_0} = \vec{V_2}$
3. $\neg(c \wedge C_1 \wedge \ldots C_p) \longrightarrow (\neg c \wedge \vec{V_0} = \vec{V_2})$
4. $\neg(\neg c \wedge \vec{V_0} = \vec{V_2}) \longrightarrow (c \wedge C_1 \wedge \ldots \wedge C_p \wedge w(c,\vec{V_1},\vec{V_3},\vec{V_2},C_1 \wedge \ldots \wedge C_p))$

où $\vec{V_0}$,$\vec{V_1}$,$\vec{V_2}$ *et* $\vec{V_3}$ *sont des vecteurs de FD-variables.*

Les deux premières contraintes gardées sont l'expression directe du comportement d'une instruction "*while*". Lorsque la condition c est vérifiée, le corps de l'opérateur est exécuté et un nouvel appel à *w/5* est engendré, après la création de nouvelles instances de variables. Lorsque la condition est fausse, le corps n'est pas évalué et les variables d'entrée sont contraintes à être égales au vecteur des variables utilisées.

Les troisième et quatrième règles sont, comme celles du "*if*", basées sur le principe de réfutation.

La troisième contrainte gardée résulte de l'observation suivante : s'il s'avère que les contraintes du corps de l'instruction `while` sont inconsistantes par rapport à celles du système de contraintes déjà généré σ, le corps du `while` ne peut être exécuté car sa condition d'arrêt est satisfaite. La dernière contrainte gardée résulte du fait que si la valeur d'une variable n'est pas la même avant et après l'instruction "*while*", alors le corps de l'instruction `while` est exécuté au moins une fois. Ces deux contraintes ne sont pas nécessaires pour exprimer le comportement de la boucle mais elles facilitent le travail d'inversion nécessaire à la découverte des cas de test en exploitant au mieux les relations entre les variables qui apparaissent dans l'instruction `while`.

Dans l'exemple de la figure 3, l'instruction "*while_do*" est transformée en : `w`$(J_3 * U_3 \leq 16,[J_1],[J_2],[J_3],J_2 = J_3 + 1)$. La quatrième contrainte gardée associée à `w/5` (cf. proposition 3.2) est activée dans le cas où σ contient $J_1 = J_3 \wedge U_3 = 2$ car dans ce cas $\neg(\neg(J_1 * U_3 \leq 16) \wedge J_1 = J_3)$ est impliqué par les contraintes de σ. Les contraintes suivantes seront alors ajoutées à σ : $J_3 * U_3 \leq 16 \wedge J_2 = J_1 + 1 \wedge$ `w`$(J_3 * U_3 \leq 16,[J_2],[J_\#],[J_3],J_\# = J_3 + 1)$ où $J_\#$ est une nouvelle variable créée par `w/5`.

3.2.5. *Appel de procédure*

Un appel de procédure est transformé en un but à résoudre. Par exemple, une instruction telle que : $v = foo(x,29)$ devient `foo`$([X,29],[],Liste\_of\_locals,[],V)$, où `foo` est le nom de la clause générée pour la procédure foo et $Liste\_of\_locals$ est la liste des FD_variables associées aux variables locales et référencées dans les décisions de la procédure. Ce mécanisme autorise le traitement des procédures récursives.

3.3. *Génération du but*

Les décisions qui doivent être vérifiées pour atteindre un point donné de la procédure sont appelées les *dépendances de contrôle* [FER 87]. Elles peuvent être déterminées syntaxiquement dans les procédures structurées et sont en général directement transformées en contraintes.

Ainsi, les *dépendances de contrôle* associées au point 10 de la procédure foo (figure 1) sont : $C(foo,10) = (Z \leq 8) \wedge (U_3 \leq X) \wedge (T_2 \leq 20)$ et la requête CLP afin de contraindre l'exécution du programme à passer par le point 10 est :

$$:- \ C(foo,10),foo([X,Y],[],[X,Z_1,U_3,T_2],[],RET) \quad (1)$$

Le programme CLP complet et le but associé au point 10 sont donnés dans la figure 4.

```
% Programme
foo([X,Y],[],[X,Z1,U3,T2],[],RET)  :-
   X,Y,Z,T1,T2,U1,U2,U3,RET ∈ 0..2^32 − 1,
   Z = X ∗ Y,
   T1 = 2 ∗ X,
   ite(X < 4,U1 = 10 ∧ U3 = U1,U2 = 2 ∧ U3 = U2),
   ite(Z ≤ 8,
      ite(U3 ≤ X,T2 = T1 − Y ∧
         ite(T2 ≤ 20,
            ...
   RET = ...


% Requête
:- (Z ≤ 8) ∧ (U3 ≤ X) ∧ (T2 ≤ 20),foo([X,Y],[],[X,Z1,U3,T2],[],RET)
```

Figure 4. *Programme CLP engendré pour la procédure foo et requête associée*

Toutefois, pour les points situés dans le corps d'une instruction itérative, les conditions d'exécution devront être générées dynamiquement lors de la résolution du système de contraintes.

4. Résolution du but

La résolution d'un programme CLP s'effectue grâce à l'alternance des deux opérations suivantes :

1. le filtrage, utilisant les techniques habituelles de consistances partielles et de traitement de l'implication,

2. l'énumération, qui est une procédure de recherche dans l'espace des valeurs des variables.

La résolution du problème de génération de cas de tests nécessite, en général, de choisir des valeurs particulières pour le traitement de l'implication et celui des contraintes gardées et bien entendu de développer des heuristiques d'énumération ; en

effet, comme dans la plupart des CSPs, le filtrage seul ne permet pas de trouver les solutions.

4.1. *Traitement de l'implication*

Les trois approximations, présentées au chapitre 2, permettent de détecter les implications. Dans notre contexte, *l'abs–implication* s'avère souvent plus efficace. Ainsi, dans l'exemple suivant où $\sigma = (X \in 1..100) \wedge (Y \in 9..11) \wedge (X \neq Y)$, la contrainte $(X * Y \neq 100)$ n'est ni intervalle–impliquée, ni ac–impliquée car les valeurs $(X = 10, Y = 10)$ ne peuvent être retirées des domaines par la consistance d'arc ou la consistance d'intervalles. L'abs-implication conduit à ajouter la négation de $(X * Y \neq 100)$, soit $\sigma = (X \in 1..100) \wedge (Y \in 9..11) \wedge (X \neq Y) \wedge (X * Y = 100)$. Le filtrage par intervalle-consistance de σ réduit les domaines de X et Y à vide ce qui prouve que $(X * Y \neq 100)$ est impliqué par σ.

Cette opération de relaxation de la preuve d'implication peut s'interpréter comme une preuve par réfutation.

4.2. *Traitement des contraintes gardées*

Une contrainte gardée de la forme $C_1 \rightarrow C_2$ est évaluée itérativement. L'algorithme décrit dans la figure 5 définit le processus d'élimination des contraintes gardées.

/* Soit $C_1 \longrightarrow C_2$, une contrainte gardée et σ, l'ensemble courant de contraintes */	
Si	le filtrage de $\sigma \wedge \neg C_1$ par consistance d'intervalle produit une inconsistance
alors	$\sigma \leftarrow (\sigma \cup \{C_2\}) \setminus \{C_1 \longrightarrow C_2\}$ /* C_1 est *abs–impliquée* par σ */
Si	le filtrage de $\sigma \wedge C_1$ par consistance d'intervalle produit une inconsistance
alors	$\sigma \leftarrow \sigma \setminus \{C_1 \longrightarrow C_2\}$ /* $\neg C_1$ est *abs–impliquée* par σ */
Si	ni C_1 ni $\neg C_1$ ne sont pas abs–impliquées par σ
alors	**continue** /* La contrainte gardée $C_1 \longrightarrow C_2$ est suspendue dans σ */

Figure 5. *Algorithme de traitement des contraintes gardées*

Les deux premières règles permettent d'éliminer une contrainte gardée ; la troisième spécifie le cas où une contrainte reste suspendue.

La première règle permet d'ajouter C_2 à σ si C_1 est abs-impliquée par l'ensemble des contraintes de σ.

La seconde règle consiste à retirer la contrainte si la négation de la garde est impliquée par l'ensemble des contraintes de σ. En effet, celle-ci ne pourra plus être vérifiée dans le contexte de l'arbre de recherche courant et peut être retirée de σ.

Le traitement de cette règle peut être retardé jusqu'à la fin du traitement de l'ensemble des contraintes gardées car elle n'apporte pas de modification à l'ensemble des contraintes de σ.

Le traitement simultané de plusieurs contraintes gardées est à l'origine de deux difficultés :

– σ peut contenir d'autres contraintes gardées qui pourront être activées en cascade lors du filtrage ;

– σ peut contenir une contrainte composée qui ne termine pas (c'est le cas de la contrainte *w/5* qui peut engendrer récursivement un ensemble infini de nouvelles contraintes gardées). Ce problème peut être vu comme une conséquence de l'indécidabilité de l'arrêt d'un programme.

Pratiquement, ce problème sera résolu en traitant les contraintes gardées indépendamment les unes des autres. D'autres politiques de réveil existent [GOT 00a] mais ne seront pas discutées ici.

4.3. *Heuristiques d'énumération*

Le traitement que nous avons appliqué aux contraintes n'a jusqu'à présent engendré aucun point de choix. Les disjonctions introduites par les contraintes composées (contrainte *élément* par exemple dans le cas des tableaux) ont été traitées telles quelles grâce au test d'implication et au filtrage par les différentes consistances partielles présentées. En général le filtrage ne permet que de réduire le domaine des variables et ne permet pas d'obtenir un cas de test (c'est-à-dire une solution au système de contraintes). Il est donc nécessaire de mettre en œuvre un processus de recherche et d'énumération. L'énumération permet de fixer la valeur d'une variable ; choix dont les conséquences seront propagées sur le domaine des autres variables par le filtrage. Nous avons expérimenté différentes heuristiques d'énumération. Celles qui ont permis d'obtenir les meilleurs résultats reposent sur une décomposition par dichotomie du domaine des variables [GOT 00a].

Lorsque le processus de résolution se termine, deux cas peuvent se présenter :

– Une solution est trouvée. Dans ce cas nous avons la certitude que l'exécution du test pour les valeurs de la solution conduira le programme à traverser le point n choisi.

– Aucune solution n'est trouvée après le filtrage initial ou une énumération complète. Il est alors prouvé que le point n est un point inatteignable (code mort). Ce résultat va au-delà de la simple recherche de couverture du test et est en soi un résultat de test à proprement parler. Il est en général important de prendre conscience de la présence de code mort dans un programme.

Il est toutefois possible que la recherche ne se termine pas et que le processus doive être arrêté prématurément. Cela peut être la conséquence de la non-terminaison de l'opérateur *w/5* en présence de boucle infinie pour certaines valeurs des paramètres du programme à tester. Ce pourrait être également une information de test intéressante.

Malheureusement, rien ne permet de dire si la poursuite de la recherche finirait ou non par conduire à une solution.

5. Extensions

Dans cette section nous présentons rapidement les travaux en cours sur deux points :

– le traitement des pointeurs ;

– la prise en compte des spécificités des nombres à virgule flottante.

Il s'agit là de problèmes difficiles qui se rencontrent dans de très nombreuses applications.

5.1. *Traitement des pointeurs*

Les langages de programmation impératifs offrent souvent la possibilité d'utiliser les adresses de la mémoire vive de la machine pour accéder au contenu de certaines variables d'un programme. Dans le langage C, ces adresses, appelées pointeurs, peuvent elles-mêmes être manipulées comme des variables ; c'est-à-dire des objets dont la valeur peut être initialisée avec l'adresse de n'importe quel objet du programme créé statiquement ou dynamiquement, et modifiée en cours d'exécution.

En un certain point d'une fonction C, un pointeur et une variable peuvent désigner le même emplacement mémoire : ces deux objets sont alors synonymes. Lors de l'exécution de la fonction, une définition de l'un sera également une définition de l'autre, sans que le code source du programme n'en fasse état. Afin d'illustrer ce problème sous l'angle de la génération automatique de cas de test, considérons le problème qui consiste à générer un cas de test qui atteint l'instruction 3 dans le code de la figure 6.

```
1.    *p = 0 ;
2.    if ( i > 1 )
3.          ...
```

Figure 6. *Problème de synonymes induits par les pointeurs*

Si on suppose que p pointe vers i lors de l'exécution de l'instruction 1, alors i prend la valeur 0, ce qui rend l'instruction 3 non -exécutable ; à l'inverse, si on suppose, de manière excessivement prudente, que l'instruction 1 modifie n'importe quelle variable du programme, alors la valeur de i dans la décision 2 reste indéterminée. Ainsi, dans le premier cas, la génération de cas de test engendre un résultat éventuellement faux, tandis que le deuxième cas met en échec la génération.

Dans [GOT 00a], nous avons proposé une extension de notre méthode de génération automatique de cas de test qui prend en compte une partie de ce problème. L'idée

principale sur laquelle repose cette extension consiste à précalculer les éventuelles relations de pointage, existantes lors de l'exécution d'un point dans le programme. La connaissance de ces relations permet ainsi de prendre en compte les définitions des variables qui sont cachées par l'utilisation de synonymes. Néanmoins, ce précalcul est limité aux pointeurs vers des zones nommées de la mémoire car il s'appuie sur une analyse statique du programme. Or, le problème des synonymes existe également pour les pointeurs vers des zones anonymes de la mémoire telles que celles qui sont référencées par une allocation dynamique. Ces pointeurs sont largement utilisés dans la construction de structures dynamiques. L'extension de INKA pour le traitement de ces pointeurs est actuellement en cours d'étude.

5.2. *Extension de* INKA *aux nombres à virgule flottante*

Un des points les plus délicats dans la génération automatique de cas de tests est le traitement des nombres à virgule flottante. Les difficultés proviennent essentiellement de l'arithmétique des nombres à virgule flottante. La pauvreté des propriétés mathématiques de ces arithmétiques se traduit par une plus grande sensibilité aux problèmes sous-jacents de l'ATDG. En particulier, la correspondance entre le code source testé, qui sert de base à la génération des contraintes, et le code objet exécuté est délicate à établir. Par exemple, chaque bibliothèque mathématique a ses propres spécificités dont INKA doit tenir compte. Plus généralement, le résultat de l'évaluation d'une expression $e(x_1, \ldots, x_n)$ pour un n-uplet de nombres à virgule flottante dépend de divers paramètres (e.g., le mode d'arrondi, la bibliothèque mathématique, l'unité de calcul en virgule flottante).

Les contraintes sur les flottants étant définies par des expressions arithmétiques, l'utilisation directe des solveurs sur les réels [8] pourrait être envisagée pour résoudre des contraintes sur les flottants. L'utilisation directe de tels solveurs se heurte toutefois à deux problèmes majeurs :

– *Les solveurs sur les réels ne sont pas conservatifs sur les flottants* i.e., ils peuvent supprimer des solutions sur les flottants qui ne sont pas des solutions sur les réels :

- Par exemple, l'équation $16.0 = 16.0 + x$ est considérée par un solveur basé sur un filtrage par $2B$–consistance (e.g., `PROLOG IV` [COL 94]) comme étant équivalente à $x = 16.0 - 16.0$ qui produit comme résultat $x = 0$ alors que, sur les flottants, cette équation admet pour solution n'importe quel flottant dans l'intervalle $[-8.88178419700125232e-16, 1.77635683940025046e-15]$.

- un solveur basé sur un filtrage par Box–consistance (e.g., `Numerica` [VAN 97] ou `DeClic` [GUA 00]) peut retirer des solutions sur les flottants à cause des manipulations dues à la forme de Taylor utilisée par l'extension aux intervalles de la méthode de Newton. Par exemple, considérons l'équation $f(x,y,z) = x + y + z = 0$

8. Les contraintes sur les flottants sont certes des contraintes sur les domaines finis mais la taille des domaines exclut des techniques classiques des CSP (il y a plus de 10^{18} nombres à virgule flottante dans l'intervalle $[-1, 1]$ lorsque les flottants sont représentés par des `double`.)

avec $x \in X = [-1, 1]$, $y \in Y = [16.0, 16.0]$ et $z \in Z = [-16.0, -16.0]$. Une unique itération de l'opérateur de Newton sur les intervalles, $X := X \cap (m(X) - \frac{f(m(X),Y,Z)}{\frac{\partial f}{\partial x}(X,Y,Z)})$, retourne immédiatement $X = [0, 0]$, alors que l'ensemble des solutions sur les nombres à virgule flottante est beaucoup plus important.

Ces problèmes sont évidement amplifiés par les transformations symboliques effectuées par certains solveurs pour mieux réduire les intervalles.

– Les solutions fournies par les solveurs sur les réels sont de petits intervalles qui peuvent contenir une solution sur les réels alors que les solutions recherchées sur les flottants sont des n-uplets de nombres à virgule flottante. Ainsi, `PROLOG IV` nous indique que des solutions de l'équation $x^2 = 2$ peuvent être contenues dans l'intervalle $]1.4142135,1.4142136[$.

Ces exemples montrent qu'il n'est pas possible d'utiliser directement un solveur sur les réels. Il est donc nécessaire d'introduire un nouveau solveur basé sur un algorithme de filtrage conservatif i.e., un algorithme qui ne supprime aucune solution sur les flottants et capable d'énumérer sur les flottants.

On trouvera dans [MIC 01] une ébauche d'un tel solveur. Schématiquement, celui-ci est basé sur les principes suivants :

– Une définition d'un véritable système de contraintes sur les flottants avec notamment un mode d'évaluation des contraintes qui tient compte du mode d'arrondi et qui correspond à celui des expressions du langage C respectant le standard ANSI C ;

– Un algorithme de filtrage conservatif qui permet de réduire le domaine des variables sans perdre aucune solution sur les flottants ; cet algorithme utilise en particulier des techniques de consistance partielle telles que la 2B-consistance ou la Box-consistance comme heuristique de choix ;

– Un algorithme d'énumération qui permet de générer des n-uplets de flottants.

6. Résultats expérimentaux

Nous avons tenté de comparer expérimentalement notre approche avec celle de la génération de test aléatoire d'une part, et l'approche dynamique de [KOR 90, FER 96] d'autre part. Pour cela, nous avons implémenté la méthode aléatoire en utilisant la fonction `drand48` de la bibliothèque standard de C qui permet d'engendrer une série de nombre pseudo-aléatoires. `TESTGEN`, une implémentation de l'approche dynamique pour le langage Pascal, n'étant pas disponible, nous avons basé notre comparaison sur les résultats publiés par R. Ferguson et B. Korel dans [FER 96]. Nous aurions également souhaité comparer notre approche avec les méthodes d'exécution symbolique mais, bien qu'un outil existe et soit disponible [DEM 93], il s'est avéré inutilisable pour notre comparaison dans la mesure où il est conçu pour l'analyse des mutations des programmes Fortran.

6.1. *Notre prototype*

Nous avons réalisé INKA, un prototype de notre approche opérant sur un sous-ensemble du langage C. Ce sous-ensemble exclut les instructions `goto`, l'arithmétique sur les pointeurs, les structures de données dynamiques, les pointeurs de fonctions, et le forçage de type. Les types de données pris en compte sont les entiers (`char`, `short`, `long`, `...`) et presque tous les opérateurs du langage (34 parmi 42) grâce à des contraintes spécifiques que nous avons définies.

INKA comporte un analyseur du langage C, un générateur de forme SSA et la génération des contraintes dans le langage de la bibliothèque CLP(FD) de Sicstus Prolog, langage avec lequel l'ensemble des programmes ont été réalisés.

6.2. *Expérimentations*

Nous avons essayé notre approche sur un ensemble de programmes [GOT 00a]. Nous présentons notre comparaison sur trois programmes [9] habituellement considérés par la communauté "Test de logiciel" : 1) "bsearch" [DEM 93] recherche binaire dans un arbre trié, 2) "sample", un programme publié dans [FER 96] qui contient des tableaux, des itérations et un grand nombre de dépendances de contrôle, 3) "trityp" [DEM 93] célèbre par le grand nombre de chemins non exécutables qu'il contient.

Nous présentons également des résultats pour un programme extrait d'un logiciel avionique opérationnel que nous appellerons "avion". Il contient de nombreux tests imbriqués, des opérations bit à bit mais pas d'instructions itératives.

6.3. *Procédure de test*

Nous avons engendré un cas de test pour chaque bloc de base (séquence d'instructions sans branchement) du graphe de contrôle du programme à tester. Bien entendu il n'est pas nécessaire de considérer tous ces cas pour assurer 100% de la couverture des branches, mais ce qui nous intéresse ici est la capacité de notre approche à déterminer les données de test.

L'expérimentation a été conduite sur une station Sun UltraSparc 5 [10]. Une limite de temps de 10 secondes a été fixée pour chaque cas de test. Durant ce laps de temps, la méthode aléatoire engendrait environ 10^5 cas de test, alors qu'INKA n'en considérait qu'un. Pour réduire les hasards de la méthode aléatoire, nous avons répété chaque génération 10 fois avec des germes de départ différentes et retenu que les meilleurs résultats (en terme de couverture).

---

9. Le code source de ces programmes est disponible à l'adresse :
http://www.essi.fr/~rueher/trityp.htm
10. Opérant à 300 Mhz

[FER 96] décrit -entre autres- les résultats obtenus par TESTGEN (approche dynamique) sur les trois programmes retenus. Les expérimentations ont été conduites sur un processeur Pentium à 60Mhz. Le time-out était fixé à 5 minutes et la recherche (répétée 10 fois) portait -comme dans notre cas- sur chacun des blocs du programme. Le taux de couverture ainsi calculé représentait le pourcentage de cas pour lesquels un essai au moins permettait d'engendrer des cas de test valides [FER 96]. Selon cette définition, la couverture atteinte fut de 100% dans les trois cas.

6.4. *Résultats*

La figure 7 présente l'ensemble des résultats. Les deux premières colonnes du tableau indiquent le nombre de lignes de code et le nombre de blocs ; la troisième représente une estimation de la taille de l'espace de recherche (nombre total de cas de test possibles). Les trois dernières colonnes fournissent les taux de couverture obtenus pour chacune des approches.

Programs	**loc**	**blocks**	**test data**	TESTGEN*	**Random****	INKA**
bsearch	21	10	$> 10^{50}$	100%	100%	100%
sample	33	14	$> 10^{100}$	100%	93%	100%
trityp	40	22	$> 10^{10}$	100%	86%	100%
avion	157	38	$> 10^{60}$	–	74%	100%

(*) 50 minutes sur PC Pentium (60Mhz) pour chaque bloc (**) 10 secondes sur Sun Sparc 5 (300Mhz) sous Solaris 2.5 pour chaque bloc

Figure 7. *Comparaison du taux de couverture*

6.5. *Analyse des résultats*

Même si les comparaisons précises sont difficiles à faire en raison de la différence des machines utilisées, il apparaît clairement que INKA est environ 10 fois plus rapide que Testgen [11].

Nous avons analysé en détail les résultats obtenus pour le programme "Trityp". La Figure 8 représente le temps d'exécution requis pour chacun des blocs. On notera que la méthode aléatoire peut être plus efficace qu'INKA sur certains blocs (en raison du faible surcoût requis par l'approche) mais qu'elle échoue complètement pour certains blocs (par exemple le bloc 14 qui requiert le choix de 3 entiers égaux, cas hautement improbable). Ce cas est au contraire très facile pour INKA car ces valeurs sont une conséquence directe des contraintes posées.

11. On remarquera également que INKA est écrit en Prolog tandis que TESTGEN l'est en C.

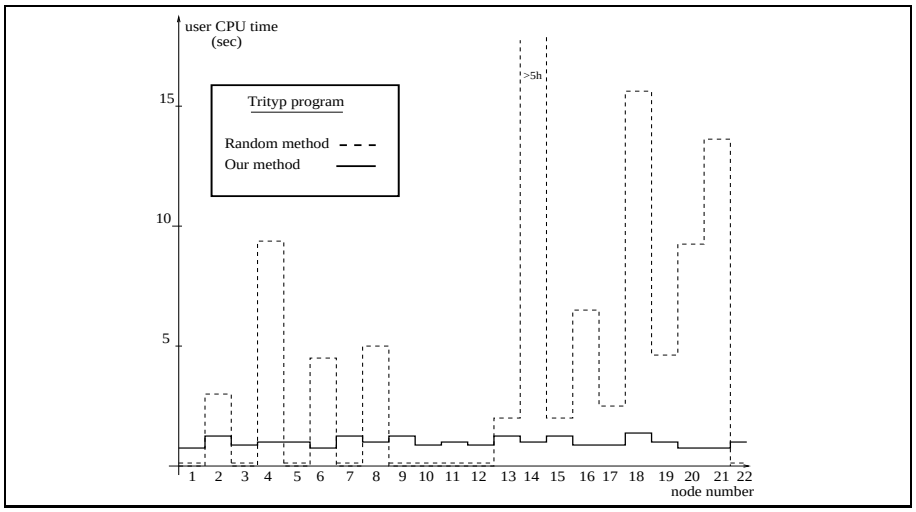

Figure 8. *Temps requis pour engendrer une solution pour chaque bloc.*

7. Conclusion

Le système INKA que nous avons présenté dans cet article repose sur une nouvelle méthode pour la génération de cas de test structurels : la modélisation du programme à tester par un système de contraintes. Cette modélisation offre plus de déclarativité et autorise la génération d'un cas de test à partir du seul programme et de l'instruction sélectionnée ; en particulier aucun chemin n'est choisi a priori (ce qui facilite la détection de code mort dans un certain nombre de cas). Le prototype expérimental que nous avons réalisé a permis de valider cette méthode. Les travaux futurs portent à la fois sur l'extension du système INKA pour autoriser le traitement d'une classe de programmes plus importante et le transfert industriel. Le premier point concerne en particulier un traitement plus complet des pointeurs, une résolution correcte et efficace des contraintes sur les flottants, et une prise en compte de certaines constructions d'un langage comme C++. Le transfert industriel s'effectue dans le cadre d'un projet "pré-compétitif RNTL" dont l'objectif est la réalisation d'un outil opérationnel pour la génération de cas de test structurels à partir du seul code source de programmes écrits en C et C++. Une première version de INKA sera disponible fin 2002. Outre `THALES SYSTEMES AEROPORTES`, ce projet implique la société `AXLOG` et les laboratoires Universitaires *I3S-CNRS*, *LIFC* et *LSR-IMAG*.

8. Bibliographie

[AER 98] AERTRYCK L. V., « Une méthode et un outil pour l'aide à la génération de jeux de tests logiciels », PhD thesis, Université de Rennes I, 1998.

[AHO 86] AHO A., SETHI R., ULLMAN J., *Compilers Principles, techniques and tools*, Addison-Wesley Publishing Company, 1986.

[ARN 99] ARNOULD A., MARRE B., GALL P. L., « Génération automatique de tests à partir de spécifications de structures de données bornées », *Technique et Science Informatiques*, vol. 18, n° 3, 1999, p. 297–321.

[BOU 99] DU BOUSQUET L., OUABDESSELAM F., RICHIER J.-L., ZUANON N., « L utess: a Specification-driven Testing Environment for Synchronous Software », *21st International Conference on Software Engineering*, Los Angeles, May 1999.

[CAR 94] CARLSON B., CARLSSON M., DIAZ D., « Entailment of Finite Domain Constraints », *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, MIT Press, 1994, p. 339-353.

[COL 94] COLMERAUER A., « Spécifications de Prolog IV », rapport, 1994, GIA, Faculté des Sciences de Luminy,163, Avenue de Luminy 13288 Marseille cedex 9 (France).

[CYT 91] CYTRON R., FERRANTE J., ROSEN B. K., WEGMAN M. N., ZADECK F. K., « Efficently Computing Static Single Assignment Form and the Control Dependence Graph », *Transactions on Programming Languages and Systems*, vol. 13, n° 4, 1991, p. 451-490.

[DEM 93] DEMILLO R. A., OFFUT A. J., « Experimental Results from an Automatic Test Case Generator », *Transactions on Software Engineering Methodology*, vol. 2, n° 2, 1993, p. 109-175.

[FER 87] FERRANTE J., OTTENSTEIN K. J., WARREN J. D., « The Program Dependence Graph and its use in optimization », *Transactions on Programming Languages and Systems*, vol. 9-3, 1987, p. 319-349.

[FER 96] FERGUSON R., KOREL B., « The Chaining Approach for Software Test Data Generation" », *ACM Transactions on Software Engineering and Methodology*, vol. 5, n° 1, 1996, p. 63-86.

[GOT 98] GOTLIEB A., BOTELLA B., RUEHER M., « Automatic Test Data Generation Using Constraint Solving Techniques », *Proc. of the Sigsoft International Symposium on Software Testing and Analysis*, Clearwater Beach, Florida, USA, March 2-5 1998, *Software Engineering Notes*,23(2):53-62, available at http://www.essi.fr/ rueher/.

[GOT 00a] GOTLIEB A., « Automatic Test Data Generation using Constraint Logic Programming », PhD thesis, PHD Dissertation (in French), Université de Nice–Sophia Antipolis, January 2000.

[GOT 00b] GOTLIEB A., BOTELLA B., RUEHER M., « A CLP Framework for Computing Structural Test Data », *Proc. of CL2000*, LNAI 1891, Springer Verlag, July 2000, p. 399–413.

[GUA 00] GUALARD F., « Langages et environnements en programmation par contraintes d'intervalles », PhD thesis, Université de Nantes — 2, rue de la Houssinière, F-44322 NANTES CEDEX 3, France, 2000.

[GUP 99] GUPTA N., MATHUR A. P., SOFFA M. L., « UNA Based Iterative Test Data Generation and its Evaluation », *Proc. of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach, Florida, USA, October 1999.

[GUP 00] GUPTA N., MATHUR A. P., SOFFA M. L., « Generating Test Data for Branch Coverage », *Proc. of 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble , France, September 2000.

[HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILLAUD D., « The synchronous data flow programming language LUSTRE », *Proc. of IEEE*, vol. 79, September 1991, p. 1305–1320.

[HEN 98] HENTENRYCK P. V., SARASWAT V., DEVILLE Y., « Design, Implementation, and Evaluation of the Constraint Language cc(FD) », *Journal of Logic Programming*, vol. 37, 1998, p. 139-164, Also in CS-93-02 Brown–University 1993.

[JAF 86] JAFFAR J., LASSEZ J.-L., MAHER M., « A Logic Programming Language Scheme », DEGROOT D., LINDSTROM G., Eds., *Logic Programming: Relations, Functions and Equations*, p. 441–468, Prentice Hall, 1986.

[JAF 94] JAFFAR J., MAHER M., « Constraint Logic Programming: A Survey », *Journal of Logic Programming*, vol. 19/20, 1994, p. 503–581.

[KIN 76] KING J. C., « Symbolic Execution and Program Testing », *CACM*, vol. 19, n° 7, 1976, p. 385-394.

[KOR 90] KOREL B., « Automated Software Test Data Generation », *IEEE Transactions on Software Engineering*, vol. 16, n° 8, 1990, p. 870-879.

[LIO 97] LIONEL VAN AERTRYCK MARC BENVENISTE D. L. M., « CASTING: A formally based software test generation method », *1st International Conference on Formal Engineering Methods, IEEE, ICFEM'97, Hiroshima, Japan*, 1997.

[LöT 00a] LÖTZBEYER H., PRETSCHNER A., « AutoFocus on Constraint Logic Programming », *Proc. of (Constraint) Logic Programming and Software Engineering (LPSE'2000), London*, 2000.

[LöT 00b] LÖTZBEYER H., PRETSCHNER A., « Testing Concurrent Reactive Systems with Constraint Logic Programming », *Proc. of 2nd workshop on Rule-Based Constraint Reasoning and Programming, Singapore*, 2000.

[MAC 77] MACKWORTH A., « Consistency in networks of relations », *Journal of Artificial Intelligence*, , 1977, p. 8(1):99–118.

[MAR 00] MARRE B., ARNOULD A., « Test sequences generation from lustre descriptions: Gatel », *Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000*, Grenoble, Septembre 2000, p. 229–237.

[MEU 98] MEUDEC C., « Automatic Generation of Software Test Cases from Formal Specifications », PhD thesis, Queen's University of Belfast, 1998.

[MIC 01] MICHEL C., RUEHER M., LEBBAH Y., « Solving constraints over floating-point numbers », *Research Report No 2001-11*, I3S/CNRS,Université de Nice–Sophia Antipolis, Mars 2001.

[NTA 98] NTAFOS S., « On Random and Partition Testing », *Proceedings of Sigsoft International Symposium on Software Testing and Analysis*, vol. 23(2), Clearwater Beach, FL, March,2-5 1998, ACM, SIGPLAN Notices on Software Engineering, p. 42-48.

[PAR 99] PARGAS R., HARROLD M. J., PECK R., « Test-Data Generation Using Genetic Algorithms. Journal of Software Testing, Verifications, and Reliability », *Journal of Software Testing, Verifications, and Reliability*, vol. 9, 1999, p. 263-282, Also in CS-93-02 Brown–University 1993.

[PRE 01] PRETSCHNER A., LÖTZBEYER H., « Model Based Testing with Constraint Logic Programming: First Results and Challenges », *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto*, 2001.

[VAN 97] VAN HENTENRYCK P., MICHEL L., DEVILLE Y., Eds., *Numerica : a Modeling Languge for Global Optimization*, MIT press, 1997.

[WEY 79] WEYUKER E., « Translatability And Decidability Questions For Restricted Classes Of Program Schemas », vol. 8, n° 4, 1979, p. 587-598, SIAM J. COMPUT.

[ZHU 97] ZHU H., HALL P., MAY J., « Software Unit Test Coverage and Adequacy », *Computing Surveys*, vol. 29, n° 4, 1997, p. 366–426.

Article reçu le 24 juillet 2001
Version révisée le 9 avril 2002
Rédacteur responsable : Bruno Marre

Bernard Botella *est membre de la direction technique de THALES Systèmes Aéroportés où il encadre les travaux sur l'application de techniques d'IA pour le génie logiciel. Il est actuellement responsable du projet InKa. Ses thèmes d'intérêt sont la représentation de connaissances, la programmation par contraintes, les spécifications formelles et la vérification automatique de logiciels.*

Arnaud Gotlieb *est docteur en Informatique de l'Université de Nice-Sophia Antipolis. Il a travaillé chez THALES Systèmes Aéroportés de 1995 à 2002 en tant que thésard puis responsable du projet InKa. Il est actuellement Chargé de Recherche à l'IRISA de Rennes. Ses travaux portent sur l'application des techniques de programmation par contraintes et d'analyse statique au test logiciel.*

Claude Michel *est docteur en Informatique de l'Université de Nice-Sophia-Antipolis. Ingénieur de recherche à l'université de Nice-Sophia Antipolis, il est plus particulièrement chargé de l'extension aux flottants de Inka. Ses travaux portent sur l'application des techniques issues de la programmation par contraintes à l'analyse statique de programmes.*

Michel Rueher *est Professeur à l'Université de Nice - Sophia Antipolis. Il est enseignant à l'ESSI (Ecoles Supérieure en Sciences Informatiques de l'Université de Nice - Sophia Antipolis) et effectue ses recherches dans le cadre du projet COPRIN, projet commun entre l'I3S, l'INRIA Sophia Antipolis et le CERMICS. Ses travaux de recherche portent essentiellement sur le développement et l'application de nouvelles techniques de programmation par contraintes dans le domaine continu.*

Patrick Taillibert *est responsable des activités en Intelligence Artificielle chez THALES Systèmes Aéroportés. Ses centres d'intérêts actuels concernent : la représentation de connaissances, le diagnostic et la planification automatique, la programmation à contraintes sur intervalles, les systèmes multi-agents et les algorithmes «anytime». La programmation logique à contraintes est l'un des outils privilégié par son équipe pour le développement de systèmes intelligents. C'est dans ce contexte qu'ont été menées les recherches ayant abouti au logiciel InKa.*