Utilisation de contraintes pertinentes pour la génération automatique de jeux de tests

J.F.Pradat-Peyre et J.Printz

Conservatoire National des Arts et Métiers
Laboratoire CEDRIC
292 rue St Martin
75141 Paris CEDEX 03

peyre@cnam.fr, printz@cnam.fr

Résumé

Le test de logiciel, ou de composants logiciels, représente une part importante du processus d'élaboration d'applications informatiques. Un des problème majeur de cette activité est la difficulté à produire des jeux de tests pertinents, c'est-à-dire permettant la mise en évidence d'éventuelles erreurs. Dans le cadre de l'action FORMA (DRET/CNRS/MENESR), nous étudions, en collaboration avec Dassault Electronique, la nature des contraintes à fournir aux outils de génération automatique de tests afin d'en améliorer la qualité. Nous considérons dans un premier temps des contraintes portant sur les paramètres du composant à tester. L'utilisation de telles contraintes permet en effet la production de jeux de tests faisant fonctionner le composant sur le bord de son domaine de définition, zone où se situe la majorité des erreurs. Un prototype utilisant l'outil Devisor de Dassault Electronique devrait être développé au cours de cette action.

1. Introduction

L'importance du poste test est une constante de tous projet impliquant le développement de logiciels complexes. Pour le développement des logiciels de la navette spatiale (2.2 Millions de lignes en HAL), la NASA a fait état d'un coût de l'ordre de 65% par rapport au coût global des logiciels. Ces chiffres sont corroborés par nos industriels développant des logiciels de taille et de type comparables.

La quantité de tests à fabriquer pour de tels systèmes est considérables car les logiciels associés à leur environnement système forment des ensembles combinatoires de très grande taille. Il est impossible de construire les cas de tests correspondant à une combinatoire complète en demeurant dans des structures de coûts et des délais raisonnables.

En terme de durée et d'effort, les phases où les tests sont prépondérants représentent couramment plus de 50% du total du projet. Ce sont les phases où l'on enregistre généralement le plus de retard, en particulier du fait que ces phases sont peu, ou mal, outillées. De plus il est important de savoir rejouer, sans intervention humaine, l'ensemble de tests déjà effectués pour assurer la non régression de ces logiciels qui sont tous construits de façon incrémentale.

Il faut donc disposer de techniques de fabrication de tests les plus automatisées possibles, et surtout, de données et de contraintes sur l'environnement permettant de générer des tests pertinents, susceptibles de déceler les erreurs là où elles se trouvent généralement concentrées, c'est à dire au niveau des interfaces ou des conditions aux limites.

Le but de cet article est de présenter le travail que nous effectuons dans le cadre de l'action FORMA en collaboration avec la société Dassault Electronique concernant la génération automatique

de jeux de tests. Nous concentrons notre effort sur la génération automatique de jeux de tests faisant intervenir des valeurs aux limites. Ces valeurs sont obtenues par résolution de systèmes linéaires de contraintes, définis par l'utilisateur qui utilise sa connaissance du composant à tester.

Après avoir défini les différents types de tests et leur rôle, nous présentons les types de contraintes que doit fournir l'utilisateur sur les paramètres du composant à tester, nous montrons comment ces contraintes peuvent être utilisées pour générer des jeux de tests, et enfin nous concluons et donnons quelques perspectives de ces travaux.

2. Définition et rôle des tests

Deux grandes familles de types de tests existent : les tests *boites noires* et les tests *boites blanches* [Bei89].

Dans le cas des tests ***boites noires***, on ne prend en compte que les données d'entrée et les données de sortie, en s'interdisant de regarder le contenu de l'élément à tester. C'est la validation du logiciel par rapport à sa spécification. Il faut donc analyser la structure fine des domaines en entrée et leurs correspondances avec les résultats attendus. Pour ne pas être immédiatement confronté au problème de combinatoire, il faut une stratégie de construction des tests de validation fondée sur une connaissance approfondie des modes d'emploi du logiciel. À partir d'un ensemble de scénarios aussi représentatifs que possible des besoins des usagers, on peut définir des variantes pour valider la robustesse, les performances, etc.

L'inconvénient des tests boites noires est d'abord la combinatoire qui peut conduire à des rendements de l'effort de test particulièrement bas, surtout si les scénarios sont peu représentatifs de l'usage réel du logiciel, ou si les personnes en charge de la qualification connaissent mal le sujet. L'analyse des domaines peut être quasiment impossible si l'on n'a pas un minimum d'information sur la structuration du logiciel, ce qui empêchera de valider les franchissements de frontières.

La recherche de contraintes pertinentes est donc économiquement très importante.

Dans le cas des tests ***boites blanches***, on prend en compte l'intégralité de l'élément à tester et on observe l'élément à un niveau de granularité quelconque à l'aide de l'instrumentation disponible dans le programme et/ou dans le système d'exploitation. C'est la *vérification* du logiciel par rapport à sa réalisation concrète.

Les tests *boites blanches* donnent une vue analytique plus ou moins fine du comportement d'un élément de programme. C'est là où réside la difficulté car, selon la granularité de l'observation, la quantité d'information manipulée peut devenir considérable. Les tests sont totalement dépendants de la structure de l'élément, et ce d'autant plus que l'observation est fine. L'environnement du test peut être très complexe, en particulier s'il faut simuler des éléments non intégrés.

Une bonne stratégie de tests consiste à combiner tests boites noires et tests boites blanches, en faisant varier les points de vue:

- architecture, où l'on s'intéresse surtout aux interfaces,
- programmation, où l'on raisonne en terme de couverture des instructions du programme,
- intégration, où l'on examine les enchaînements de fonctions en vue de constituer des chaînes longues de façon à estimer le *MTTF* (*Mean Time To Fail*) du logiciel,
- maintenance, où l'on va calculer le *MTTR* (*Mean Time To Repair*) du logiciel,
- l'exploitation, où l'on s'attache au comportement global et au point de vue de l'usager que l'on va caractériser à l'aide du temps de réponse du logiciel.

Une bonne métrologie est donc essentielle si l'on ne veut pas gaspiller l'effort consacré à la production de tests.

L'approche par identification des contraintes pertinentes est ~~donc~~ un bon moyen de rester concentré sur les aspects essentiels du logiciel à valider. Son couplage avec l'outil DEVISOR permet par ailleurs de récupérer tous les automatismes de l'environnement DEVISOR. La combinaison des deux devrait donc améliorer la productivité de l'effort de test.

3. Description des contraintes et de leur utilisation

Dans une démarche classique de tests, le testeur (c'est à dire la personne conduisant les tests) utilise sa connaissance du composant à tester pour fournir manuellement à l'outil utilisé des jeux de valeurs pertinentes. Le test effectué, cette personne doit analyser les résultats obtenus afin de vérifier s'ils sont corrects par rapport à ceux attendus. On demande donc à la personne conduisant les tests de jouer à la fois le rôle d'expert (celui qui connaît la pièce et sait fournir les bons jeux de valeurs) et le rôle d'oracle (celui qui sait interpréter les résultats produits). Ces deux activités étant consommatrices de temps, le testeur est souvent contraint de réduire le nombre de jeux de tests.

Une étude menée dans le cadre de l'action FORMA tend à montrer que les valeurs conduisant à la détection d'erreurs sont quelques fois des valeurs remarquables dans le domaine de validité des paramètres de la pièce à tester et le plus souvent des valeurs se situant aux frontières de ce domaine de validité. Les erreurs semblent provenir principalement de la non prise en compte de valeurs d'entrées anormales ou peu probables (non traitement des exceptions, oubli de tests d'appartenance au domaine de définition ou même, mauvaises bornes de boucles).

Notre objectif est de fournir une aide à la personne menant les tests à la fois dans son rôle d'expert, en automatisant la production de jeux de valeurs pertinentes, ainsi que dans son rôle d'oracle, en automatisant une partie de l'analyse des résultats. Pour ce faire, on demande au testeur de fournir non plus des jeux de valeurs mais un ensemble de contraintes liant les paramètres d'entrées et de sortie de la pièce à tester.

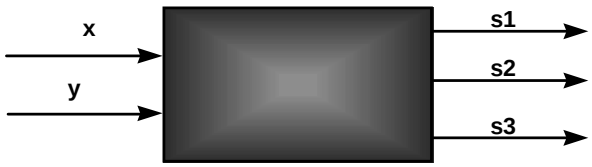
3.1. Description des contraintes

La personne chargée de conduire le test d'un composant dispose d'une connaissance plus ou moins formelle de celui ci par le biais de spécifications et/ou de documents de conception. Cette connaissance va lui permettre d'exprimer un certain nombre de contraintes relatives aux paramètres d'entrée et aux résultats produits.

Nous faisons l'hypothèse que ces contraintes sont exprimées sous une forme normale disjonctive : les contraintes sont une disjonction de cas (cas1 v cas2 v ... casn) où chaque cas est de la forme E => F avec E un système mixte de contraintes linéaires (équations ou inéquations) portant sur les paramètres d'entrée et F un système mixte de contrainte (équations ou inéquations mais non nécessairement linéaires) portant sur les paramètres de sortie. Cette expression de contraintes peut s'interpréter par si E est vérifié alors F doit être vérifié. Le système E sert donc à générer des jeux de valeurs et le système F sert à vérifier que le résultat produit est conforme à celui attendu. Dans le cas où F est vide, le résultat attendu est supposé être : rien de mauvais ne se passe durant l'exécution. Il faut noter que ce type de contraintes s'applique difficilement sur des composants de type boucle permanentes.

Exemple :

Considérons une pièce à tester comportant deux paramètres d'entrée, x et y, et trois paramètres de sortie, s1, s2 et s3. Tous ces paramètres sont de type entiers.



Un exemple de contraintes fournies par le testeur peut être :

$$\left[(x \geq 1) \Lambda \left(y \leq 2\right)\right] \text{ v } \left[x \geq y \Rightarrow s1 = x * y\right] \text{ v } \left[x < y \Rightarrow s3 = -1\right]$$

Dans le premier cas il souhaite simplement vérifier que rien de mauvais ne se passe, dans les autres cas que le résultat obtenu vérifie certaines propriétés.

Lorsque les types des paramètres sont des types simples (entier, caractère, booléens, etc.) il n'y a pas de problèmes particuliers pour exprimer ces contraintes. Lorsque ces types sont structurés (sommes, récursifs ou références) il est nécessaire de préciser sur quoi porteront les contraintes.

Dans le cas de type somme (enregistrement) nous faisons le choix d'exprimer les contraintes à l'aide de systèmes mixtes portant sur les champs de ce type. Par exemple, si une variable est de type complexe, les contraintes porteront sur la partie réelle et imaginaire de la variable.

Dans le cas de type récursif (liste, arbre, etc.), il devient difficile de manipuler la variable elle même. Nous faisons alors le choix de faire porter les contraintes non pas sur la variable elle même mais sur son cardinal (ou toute autre projection de ce type vers un type simple). Pour le cas d'une liste, on pourra considérer soit son cardinal soit, par exemple, le nombre de fois qu'apparaît un élément particulier.

Enfin, en ce qui concerne les références (pointeurs, référence de méthodes, etc.), nous faisons le choix de manipuler la valeur référencée en interdisant les références sur fonction, procédure ou méthode associée à un objet. Par exemple, pour le cas d 'un pointeur sur entier, les contraintes porteront sur la valeur de l'entier pointé.

3.2. Utilisations des contraintes

L'expression des contraintes étant maintenant précisée, nous montrons comment les utiliser dans le but de générer automatiquement des jeux de tests. Pour cela, sachant que les erreurs portent principalement sur des valeurs de paramètres aux limites du domaine de définition, nous allons utiliser ces contraintes pour générer des jeux de tests faisant intervenir des valeurs aux limites.

Du fait de la forme de l'expression des contraintes (disjonction de cas avec chaque cas de la forme : E=> F et E un système mixte de contraintes linéaires), chaque cas va définir moyennant quelques précautions un polyèdre caractérisant le domaine de définition du cas de test envisagé. En parcourant le bord de ce polyèdre, il va être possible de générer un ensemble de jeux de valeurs aux limites. Afin de formaliser cette démarche nous introduisons quelques définitions mathématiques. Dans ces définitions nous supposons que les données manipulées sont rationnelles.

Définitions *: Soient x et y deux vecteurs de* $\mathbb{Q}^n$ *et* α *une constante de Q.*

- *Le* ***produit scalaire*** *de x par y, noté* $x \circ y$ *, est défini par* $x \circ y = \sum_{i=1}^{n} x_i . y_i$
- *Si y est non nul ,l'ensemble* H *défini par* $H = \{x / x \circ y = \alpha\}$ *est appelé un* ***hyperplan****.*
- *Un* ***polyèdre*** *est un ensemble de* $\mathbb{Q}^n$ *borné par un ensemble fini d'hyperplans..*

Un résultat important du à Farkas [Far02] est que l'ensemble des solutions satisfaisant un système mixte de contraintes définit un polyèdre P et sert de définition implicite du polyèdre

$$\mathrm{P} = \{x / Ax = b, Cx \geq d\}$$

cette définition étant donnée en terme d'équations (lignes de la matrice A et du vecteur b) et en terme d'inégalités (lignes de la matrice C et du vecteur d).

Un polyèdre P a également une représentation duale paramétrée (appelée aussi caractérisation de Minkowski) exprimée en terme de combinaisons positives de lignes, de combinaisons convexes de sommets et de combinaisons positives de rayons extrémaux. L'intérêt de cette définition est qu'il est possible d'utiliser cette représentation afin de générer automatiquement des valeurs sur le bord du polyèdre considéré.

Il existe des procédures permettant de passer d'une représentation à l'autre en particulier de la représentation sous forme de systèmes de contraintes à la représentation parametrée. Parmi ces procédures on peut citer les méthodes dérivées de l'algorithme du simplexe [Chv83] ainsi que les méthodes basées sur l'algorithme de Motzkin [MRTT53], dont la plus connue est l'algorithme de Chernikova [Che68].

Dans notre contexte, le testeur définit un ensemble contraintes sous la forme de cas. Chaque cas définit un système linéaire de contraintes E. Afin de pouvoir être manipulé, E est modifié de la sorte que les inégalités strictes (< ou >) soient remplacées par des inégalités larges (≤ ou ≥) en opérant une translation (+/- 1 pour les entiers, les caractères, etc., et +/- ε pour les réels). De plus, tous les types utilisés sont transformés en rationnels. Par exemple une système de la forme x = TRUE, y = FALSE sera transforme en $x \geq 0$, $y < 0$.

La représentation duale parametrée du polyèdre défini par E est calculée dans $\mathbb{Q}^{\mathbf{n}}$ où n dénote le nombre de paramètres du composant testé. La complexité de ce calcul est en $O(k \lfloor n/2 \rfloor)$ ou k dénote le nombre de contraintes du cas considéré (ce qui reste praticable pour un nombre de paramètres inférieur à 10). Un parcours aléatoire des bords de ce polyèdre permet alors de déterminer un ensemble de jeux de valeurs qu'il faut alors replacer dans le type de base du paramètre considéré. Cette dernière opération fournit des jeux de valeurs se situant soit dans le cas de test considéré, soit dans un autre cas de test, soit dans aucun mais toujours au bord de validité, zone que l'on souhaite étudier.

Une implémentation de ces fonctionnalités est en cours. Cette implémentation qui utilisera des librairies de manipulations de polyèdres (en particulier les librairies développées à l'Irisa de Rennes [Wil93, QRR96]) devra permettre de :

1. saisir un ensemble de cas de tests exprimés sous la forme E => F,
2. calculer un ensemble de jeux de tests potentiels par parcours aléatoire du bord des polyèdres considérés,
3. calculer un ensemble de jeux de tests concrets en remplaçant chaque variable dans son domaine de définition de base,
4. définir pour chaque jeux de test à quel cas il appartient afin de pouvoir générer un rapport de test en fonction des résultats attendus.

L'exécution des cas de test et l'analyse des résultats se fera en interfaçant cette implémentation avec l'outil Devisor développé par Dassault Electronique. Le prototype réalisé servira à affiner le langage de description de contraintes, le volume de jeux de tests générés ainsi que le type de report présenté à l'utilisateur.

4. Conclusion et perspectives

Nous avons montré comment l'utilisation de contraintes portant sur les paramètres d'une pièce logicielle à tester permet de générer un ensemble de jeux de tests faisant fonctionner cette pièce aux limites de son domaine de définition.

Cette méthodologie, intermédiaire entre le test empirique et l'utilisation de technique formelles [BGLM93, Gau95], devrait permettre d'améliorer les outils de génération automatique de jeux de tests sans demander un investissement trop important de la part de l'utilisateur.

Les perspectives de ces travaux sont l'introduction de nouveaux types de contraintes, permettant par exemple de faire du test boite blanche, tout en restant accessible à de nombreux utilisateurs.

Bibliographie

[Bei89] B.Beizer, Software Testing Techniques, Van Nostrand Reinhold, New York, 1989

[BGLM93] G. Bernot, M.C.Gaudel, P. Le Gall, B.Marre Experience with Black-box Testing from Formal Specification, AQUIS'93, 18-20 Oct.93

[Che68] N.V.Chernikova, Algorithm for discovering the set of all the solutions of a linear programming problem. U.S.S.R. Computational Mathematics and Mathematical Physics, 8, 1968

[Chv83] V.Chvatal, Linear Programming, W.H.Freeman and Company, New York, 1983

[Far02] J.Farkas, Theorie des einfachen Ungleichungen, Journal fur die reine und andgewandte Mathematik, 124, pp.1-27, 1902

[Gau95] M.C. Gaudel Testing can be formal, too, TAPSOFT 95

[MRTT53] T.S.Motzkin, H.Raiffa, G.L.Thompson, R.M.Thrall, The Double Description Method, Theodore S.Motzkin : Selected Paper, 1953

[QRR96] P.Quinton, S.Rajopadhye, T.Risset, On Manipulating Z-polyedra, Publication Interne N°1016, IRISA, Jui.96

[Wil93] D.K.Wilde, A Library for Doing Polyedral Operations, Publication Interne N°785, IRISA, Dec.93

[Xan94] Xanthakis, Maurice, de Amescua, Houri, Griffet, Test & Contrôle des logiciels, méthodes, techniques & outils, EC2, Paris, 1994