Projet : exécution symbolique et test structurel

Pascale Le Gall

Date de retour : 28 janvier 2018 (date susceptible d'être différée)

1 Avant-propos : consignes générales

Ces consignes sont d'ordre général : vous pouvez les adapter en fonction du contexte.

— Les devoirs devront être faits par défaut en binôme (à la rigueur individuellement).

— Le rendu d'un devoir de programmation se présente sous la forme d'un répertoire compressé comprenant, à tître indicatif, l'ensemble des fichiers suivants :

Un fichier readme.txt qui liste l'ensemble des fichiers du répertoire en indiquant le rôle de chacun des fichiers ou répertoires présents ;

Le ou les fichiers de code source (par défaut, chaque fichier source contient un commentaire en-tête indiquant le rôle du fichier, l'auteur et les dates de création, et de révisions successives, et il est attendu que le code soit commenté) ;

Le ou les fichiers de données utilisées ainsi que le ou les fichiers de résultats, pour peu qu'ils présentent un intérêt ;

Un mode d'emploi simple pour exécuter les programmes, sur vos données de test (faire en sorte qu'une commande sans arguments permette d'exécuter vos programmes et de fournir les résultats assortis de quelques commentaires) ;

Un document rédigé, en général sous format pdf comprenant les informations suivantes :

- L'architecture générale du projet (à savoir l'organisation des principales fonctions, types de données, fichiers) ;
- Les principaux choix de conception effectués ;
- Les limitations adoptées dans le projet ;
- Les réponses aux questions directement incluses dans l'énoncé ;
- Des instructions claires d'utilisation du code ;
- Des éléments de validation (des cas d'utilisation - ou tests - illustrant de bon fonctionnement du projet) ;
- Des illustrations.

— L'exploitation de ressources externes (livres, internet, etc) est autorisée sous réserve d'indiquer les sources. En revanche, il est attendu que deux devoirs rendus ne soient pas des copies conformes l'un de l'autre ;

— La priorité est d'abord à la qualité de l'analyse du problème avant le passage à l'échelle. De même, par défaut, les aspects "interactions graphiques" ne sont pas prioritaires ;

— Il est attendu que le document rédigé soit concis, et en particulier non redondant avec le code.

2 Schéma général des outils de test structurel

Les outils de test structurel sont essentiellement de deux natures :

Analyse de couverture Les outils d'analyse de couverture prennent en arguments :
- un code source,
- un critère de couverture
- et un jeu de tests

et fournissent

1. le pourcentage des éléments couverts parmi les éléments désignés par le critère de couverture en argument,
2. et l'ensemble (éventuellement vide, en cas de couverture totale) des éléments non couverts

Génération de tests Les outils de génération de tests prennent en arguments :
- un code source,
- et un critère de couverture

et fournissent

1. un jeu de test qui couvre le critère de couverture donné en argument,
2. et éventuellement, un warning en cas d'insuccès pour couvrir le critère de couverture.

La figure 1 permet d'illustrer les deux outillages de test et de les différencier. Les outils d'analyse de couverture sont a priori plus simples que les outils de génération de test puisqu'ils se contentent d'examiner quels sont les éléments définis par le critère qui sont couverts par le jeu de tests donnés en argument.

Les outils de génération de jeux de tests (e.g. Pathcrawler pour le langage C, Pex pour le langage C#, ...) sont utilisés pour construire automatiquement des jeux de test satisfaisant des critères de test structurel (cf Section 4) tandis que les outils d'analyse de couverture (CodeCover, Emma, Clover pour le langage Java, gcov pour le langage C, ...) sont utilisés pour évaluer la qualité d'un jeu de test (au regard du critère de couverture considéré et du taux de couverture atteint) et éventuellement, le compléter.

3 Langage WHILE annoté

Le langage est défini par la grammaire suivante :

$$\begin{array}{rcl} c ::= & l : skip \\ | & l : X := a \\ | & (c; c) \\ | & if\ l : b\ then\ c\ else\ c \\ | & while\ l : b\ do\ c \end{array}$$

avec l identificateur choisi dans un ensemble L d'étiquettes (ou labels). Les parenthèses autour de $c; c$ pourront être omises.

Un programme est bien-formé si une même étiquette n'apparaît pas deux fois dans le programme.

On note
- $Labels(c)$ l'ensemble des étiquettes d'un programme bien-formé c,
- $Labels(c, cmd)$ avec $cmd \subseteq \{skip, assign, seq, if, while\}$ pour indiquer les étiquettes de c relatives au type de commandes concernées (avec l'abus qu'un singleton pourra être identifié à l'élément qui le constitue).
- $Var(c)$ l'ensemble des variables (X, Y) qui apparaissent dans c.
  Plus généralement, la fonction Var s'applique aux composantes (expressions booléennes, expressions arithmétiques) du programme. Par exemple $Var(X \leq 0) = \{X\}$.

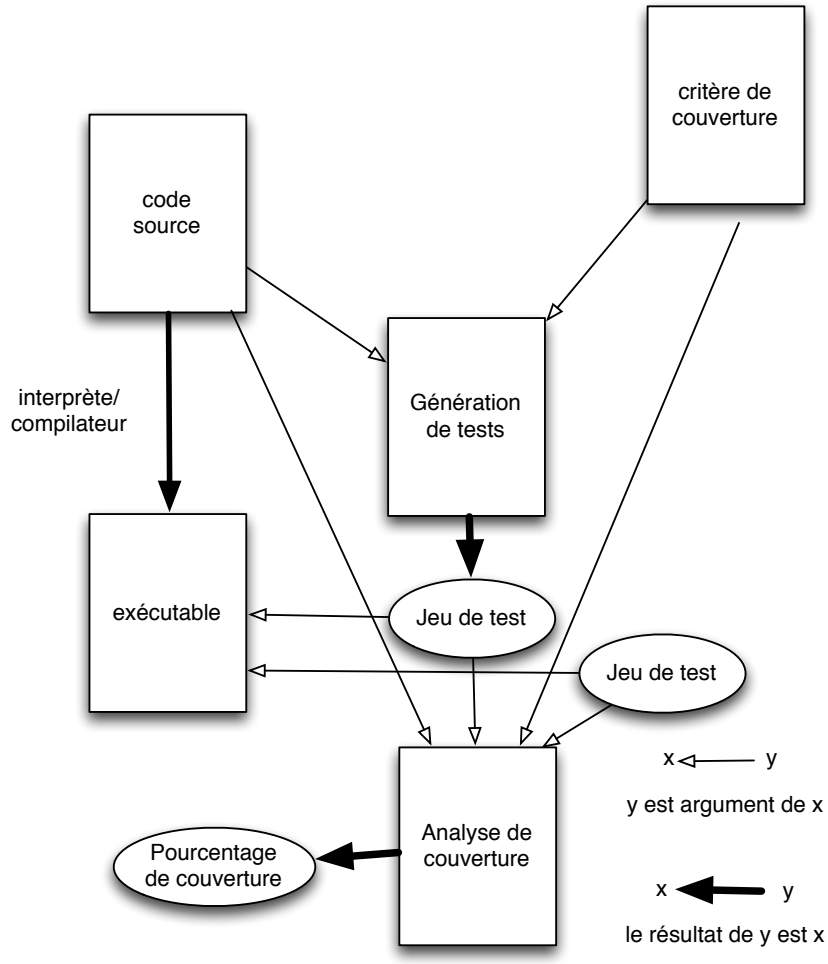

FIGURE 1 – Schéma général

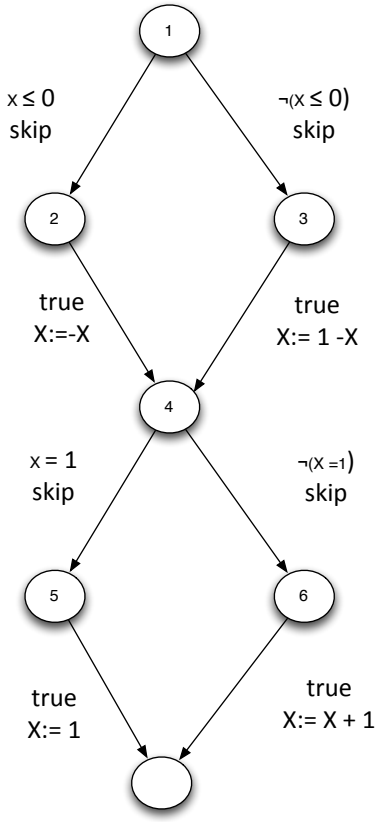

FIGURE 2 – Graphe de Contrôle de $Prog$

Soit le programme $Prog$ suivant :

$$\begin{array}{ll} 1 : if\ X \leq 0 & \\ & then\ 2 : X := -X \\ & else\ 3 : X = 1 - X; \\ 4 : if\ X = 1 & \\ & then\ 5 : X := 1 \\ & else\ 6 : X = X + 1 \end{array}$$

on a alors $Labels(Prog) = \{1, 2, 3, 4, 5, 6\}$, $Labels(Prog, assign) = \{2, 3, 5, 6\}$, $Var(Prog) = \{X\}$

Le graphe de contrôle associé à $Prog$ est donné en figure 2 (avec comme nœud d'entrée le nœud étiqueté par 1, et le nœud de sortie le nœud sans étiquette).

Un chemin du graphe de contrôle est un mot sur $Labels(Prog)$ dénotant la séquence des nœuds empruntés : 1.2.4.5._ est un chemin de $Prog$ (avec _ étiquette dénotant l'absence d'étiquettes du nœud de sortie).

Sur l'ensemble V des nœuds du graphe de contrôle, on introduit deux fonctions [1] :

1. Pour A ensemble, 2^A est l'ensemble des sous-ensembles de A.

— $def : V \to 2^{Var(Prog)}$
avec $X \in def(u)$ pour $X \in Var(Prog)$ et u nœud du graphe s'il existe une arête de la forme (u, v) étiquetée par une affectation de la forme $X = exp$
— $ref : V \to 2^{Var(Prog)}$
avec $X \in ref(u)$ pour $X \in Var(Prog)$ et u nœud du graphe s'il existe une arête de la forme (u, v) étiquetée par une expression booléenne $ExpB$ avec $X \in Var(ExpB)$ ou par une affectation de la forme $Y = ExpA$ avec $X \in Var(ExpA)$

Remarquez que par construction, s'il existe une arête (u, v) étiquetée par une expression booléenne avec $X \in Var(ExpB)$, alors il en est de même pour toutes les arêtes issues de u, et qu'il en est de même en ce qui concerne les affectations. Cela explique que ce sont les nœuds du graphe qui portent les informations de définition et utilisation des variables.

Par abus on notera $def(l)$, resp. $ref(l)$, pour $def(u)$, resp. $ref(u)$, avec u nœud étiqueté par l'étiquette l de L.

L'exécution de $Prog$ avec une donnée de test σ fournit

1. une nouvelle valuation σ' (celle définie par la sémantique opérationnelle du langage WHILE)
   On notera $exec(Prog, \sigma) = \sigma'$.
2. et un chemin ρ, défini comme un mot sur $Labels(Prog)$, (celui défini par la sémantique opérationnelle étendue pour le langage WHILE annoté).
   On notera $path(Prog, \sigma) = \rho$.

4 Jeux de test et critères de test

Une **donnée de test** pour un programme $Prog$ est définie par une valuation $\sigma : Var(Prog) \to \mathbb{Z}$.
Un **jeu de test** T pour un programme $Prog$ est un ensemble fini de données de test pour $Prog$.
Les **critères de test structurel** s'expriment en terme de couvertures des éléments du graphe de contrôle, essentiellement des étiquettes ou des arêtes entre nœuds ou encore des portions de chemins.

4.1 Critère "toutes les affectations"

Un jeu de test T pour $Prog$ satisfait le critère "toutes les affectations", dénoté TA, si toutes les étiquettes de $Labels(Prog, assign)$ apparaissent au moins une fois dans l'un des chemins d'exécution associés aux données de test σ de T.

Intuition : toutes les affectations sont exécutées au moins une fois

4.2 Critère "toutes les décisions"

Un jeu de test T pour $Prog$ satisfait le critère "toutes les décisions", dénoté TD, si toutes les arêtes (u, v) avec $Label(u) \in Labels(Prog, \{if, while\})$ sont empruntées au moins une fois dans l'un des chemins d'exécution associés aux données de test σ de T.

Remarque : il est possible qu'un chemin donné ne soit pas faisable, au sens où il n'existe pas de valuation σ vérifiant $path(Prog, \sigma) = \rho$. Dans ce cas, le chemin est retiré de facto de l'ensemble des chemins à couvrir. Cette remarque se transpose pour la plupart des critères suivants.

Intuition : tous les expressions booléennes apparaissant dans une instruction "if" ou "while" sont évaluées à vrai et à faux au moins une fois.

4.3 Critère "tous les k-chemins" (avec $k \in \mathbb{N}$)

Un jeu de test T pour $Prog$ satisfait le critère "tous les k- chemins", dénoté k-TC, si pour tous les chemins ρ de $Prog$ de longueur inférieure ou égale à k, il existe une donnée de test σ de T vérifiant $path(Prog, \sigma) = \rho$.

Intuition : tous les "petits" chemins sont exécutés au moins une fois.

4.4 Critère "toutes les i-boucles" (avec $i \in 1, 2$ en pratique)

Un jeu de test T pour $Prog$ satisfait le critère "toutes les i-boucles", dénoté i-TB, avec $i \in \mathbb{N}$ si pour tous les chemins ρ pour lesquels les boucles *while* sont exécutées au plus i fois, il existe une donnée de test σ de T vérifiant $path(Prog, \sigma) = \rho$.

4.5 Critère "toutes les définitions"

Un jeu de test T pour $Prog$ satisfait le critère "toutes les définitions", dénoté TDef, si pour toutes les variables X de $Prog$, pour tous les nœuds u de $GC(Prog)$ avec $def(u) = \{X\}$, il existe un chemin ρ de la forme $\mu_1.l_u.\mu_2.l'.\mu_3$ avec $l = Label(u)$, $X \in ref(l')$ et $\forall l \in Labels(\mu)$, $X \notin ref(l)$ pour lequel il existe une donnée de test σ de T vérifiant $path(Prog, \sigma) = \rho$.

Intuition : toutes les définitions des variables sont utilisées au moins une fois.

4.6 Critère "toutes les utilisations"

Un jeu de test T pour $Prog$ satisfait le critère "toutes les utilisations", dénoté TU, si pour toutes les variables X de $Prog$, pour tous les nœuds u de $CG(Prog)$ avec $def(u) = \{X\}$, pour tous les nœudes v $CG(Prog)$ avec $X \in ref(v)$ tel qu'il existe un chemin partiel μ de u à v, sans redéfinition de X, c'est-à-dire vérifiant $\forall l \in Labels(\mu)$, $X \notin ref(l)$, il existe un chemin ρ de la forme $\mu_1.l_u.\mu_2.l'.\mu_3$ avec $l_u = Label(u)$ et $l_v = Label(v)$, et $\forall l \in Labels(\mu_2)$, $X \notin ref(l)$. pour lequel il existe une donnée de test σ de T vérifiant $path(Prog, \sigma) = \rho$.

Intuition : toutes les utilisations accessibles par chaque définition sont exécutées au moins une fois.

4.7 Critère "tous les DU-chemins"

Pour deux nœuds u et v, on appelle *chemin simple partiel de u à v* un chemin $u.\mu.v$ de u à v qui passe au plus une fois dans chacune des boucles intermédiaires.

Un jeu de test T pour $Prog$ satisfait le critère "tous les DU-chemins", dénoté TDU, si pour toutes les variables X de $Prog$, pour tous les nœuds u de $CG(Prog)$ avec $def(u) = \{X\}$, pour tous les nœuds v $CG(Prog)$ avec $X \in ref(v)$, pour tous les chemins simples partiels μ de u à v, sans redéfinition de X, c'est-à-dire vérifiant $\forall l \in Labels(\mu)$, $X \notin ref(l)$, il existe un chemin ρ de la forme $\mu_1.l_u.\mu.l'.\mu_3$ avec $l_u = Label(u)$ et $l_v = Label(v)$, il existe une donnée de test σ de T vérifiant $path(Prog, \sigma) = \rho$.

Intuition : pour chaque couple définition-utilisation (DU) d'une variable X, tous les chemins simples sans redéfinition intermédiaire de X sont exécutés une fois.

4.8 Critère "toutes les conditions"

Les expressions booléennes utilisées dans les instructions "if" ou "while" sont appelées des *décisions*. Ces décisions peuvent être décomposées en expressions élémentaires, appelées *conditions*. Par exemple, la décision

$$(X \leq 0) \wedge (X = Y + 1)$$

est constituée des deux conditions $(X \leq 0)$ et $(X = Y + 1)$.

Un jeu de test T pour $Prog$ satisfait le critère "toutes conditions", dénoté TC, si pour toutes les conditions c de $Prog$, il existe une donnée de tests σ_c qui exécute c à vrai, et une donnée de tests $\sigma_{\neg c}$ qui exécute c à faux.

Intuition : les conditions sont exécutées avec les 2 valeurs booléennes.

4.9 Utilisation des critères

Les critères de test (TA, TD, k-TC, i-TB, TDef, TU, TDU, TC) sont utilisés avec les deux préoccupations générales suivantes :

1. pour un critère C, pour deux jeux de test T_1 et T_2 satisfaisant C avec $T_1 \subseteq T_2$, T_1 est préféré à T_2 (par souci de réduction du coût de la phase de test) ;
2. pour C_1 et C_2 deux critères de test, C_1 est dit *plus fort que* C_2 si tout jeu de tests T satisfaisant C_1, satisfait aussi C_2. Selon le contexte (en termes de criticité du logiciel sous test), les critères de test sont privilégiés : plus le logiciel est critique, plus les critères de test forts sont choisis.

Les critères de test jouent le rôle de *contrat* entre les clients et les concepteurs de logiciels. Attester qu'un logiciel vérifie tel critère est un gage de qualité.

Les critères ci-dessus sont définis sur le graphe de contrôle lorsqu'ils sont construits pour couvrir es éléments du graphe, et sur le flot de données lorsqu'ils sont construits pour couvrir des définitions utilisations de variables. Ils peuvent être utilisés seuls ou conjointement, et la littérature mentionne beaucoup d'autres critères de test structurel.

5 Travail à réaliser

Il est demandé de réaliser des outils d'analyse de couverture de test et de génération de tests pour le lanage WHILE annoté et pour les critères de test décrits en Section 4.

— Par défaut, les programmes sources (écrits en langage WHILE annoté) seront écrits dans des fichiers ".txt". Cependant, l'étape de d'analyse lexicale et syntaxique (nécessaire pour traduire le fichier source ".txt" en une structure de donnée) pourra être contournée en codant d'emblée les programmes sources dans votre structure de données.
Nota bene : selon les besoins, il est possible d'ajouter du sucre syntaxique (parenthèses, accolades, ...) et mots clés de votre choix ("and" pour $\wedge$ par exemple) afin de faciliter l'étape d'analyse lexicale et syntaxique du programme source.

— Pour les critères de test définis dans le sujet, pour un programme de votre choix, donnez des exemples de jeux de test qui satisfont (resp. ne satisfont pas) les critères en question. Indiquez quels sont les relations entre les critères (au sens de la relation "plus fort que").

Il est attendu que pour chaque critère de test, soient détaillés les mécanismes (algorithmes, caractérisations) d'analyse de couverture et de génération des jeux de test.

— La génération de tests nécessite de mettre en place la technique d'exécution symbolique afin d'associer à chaque chemin ρ son prédicat de chemin P_ρ (conjonction de contraintes portant sur les variables du programme) caractérisant les valuations σ vérifiant :

$$\sigma \models P_\rho \iff path(Prog, \sigma) = \rho$$

Par exemple $\sigma = [3 \leftarrow X, 2 \leftarrow Y]$ satisfait $P = (X \leq 5) \wedge (X = Y + 1)$, ce qui est noté $\sigma \models P$.
Les solveurs de contraintes (package constraint pour Python par exemple, choco pour java, ou des solveurs indépendants comme Yices, Z3, CVC4, ...) sont des utilitaires qui prennent en entrée des formules (dans notre cas des conjonctions d'inégalités ou d'égalités ou de leurs négations) et renvoient une valuation satisfaisant la formule en argument, s'il en existe.

(la résolution de contraintes est un problème difficile, ce qui explique que les solveurs de contraintes renvoient parfois un message pour indiquer qu'ils n'ont pas trouvé de solution dans le temps imparti).