Automatisation du test tous-les-chemins en présence d'appels de fonctions

Patricia Mouy

THÈSE

présentée à l'INSTN (Institut National de Sciences et Techniques Nucléaires)

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE D'EVRY VAL D'ESSONNE

par : Patricia Mouy

Spécialité : Informatique

Automatisation du test de tous-les-chemins en présence d'appels de fonction

Soutenue le 16 mai 2007

Composition du Jury :

– Gilles Bernot	président du jury
– Fabrice Bouquet	rapporteur
– Arnaud Gotlieb	examinateur
– Pascale Le Gall	directrice de thèse
– Yves Le Traon	rapporteur
– Bruno Marre	examinateur
– Nicky Williams	examinatrice

Table des matières

Table des figures

Liste des tableaux

Définitions et concepts

Notations

Chapitre 1

Introduction

1.1 Contexte

Le logiciel fait aujourd'hui partie intégrante de notre quotidien que se soit sur nos lieux de travail (ordinateurs, logiciels,...), à domicile (téléphone portable,...), dans nos déplacements (voiture, train,...), etc. Cette présence des logiciels dans notre quotidien accompagnée de dysfonctionnements ne nous surprend plus. Nous avons tous souffert au moins une fois d'un "plantage" intempestif de notre ordinateur ou téléphone portable pour les cas les plus courants. Ces bogues sont devenus presque habituels mais sont en général sans grande importance et sans conséquence. En revanche, si nous prenons comme exemple un défaut dans un régulateur de vitesse automobile (cas avéré il y a encore peu de temps), d'un logiciel avionique ou encore d'un logiciel de pilotage de trains, nous nous plaçons dans une logique totalement différente. La classe de logiciels concernée est toute autre. Elle regroupe les logiciels qualifiés de critiques : en général, ils interviennent dans des domaines sensibles comme ceux des transports, de la production d'énergie, de la santé et de la finance. La caractérisation commune d'un logiciel critique est tout logiciel dont une anomalie dans son fonctionnement peut avoir des conséquences beaucoup plus importantes que le bénéfice procuré par le service qu'il assure en absence d'anomalies [LAB+95]. Pour ce type de logiciels, il est primordial de pouvoir s'assurer de leurs bons fonctionnements.

Le génie logiciel permet d'acquérir la confiance désirée dans les logiciels critiques ou du moins, de s'en approcher. Selon l'arrêté du 30 décembre 1983, *"le génie logiciel est l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi"*. Myers [Mye79] définit le génie logiciel comme l'ensemble des procédures, méthodes, langages, ateliers, imposés ou préconisés par les normes adaptées à l'environnement d'utilisation afin de favoriser la production et la maintenance de composants logiciels de qualité. Le génie logiciel est basé sur des méthodologies et des outils qui permettent de formaliser et même d'automatiser partiellement la production de logiciels, mais il est également basé sur des concepts plus informels et demande des capacités de communication, d'interprétation et d'anticipation. Le génie logiciel s'intéresse à la manière dont le code source d'un logiciel est spécifié puis produit. Les différentes thématiques abordées en génie logiciel sont, de façon non exhaustive, la spécification de fonctionnalités d'un logiciel, la conception de logiciel, la génération automatique de code d'après une spécification formelle,....

Comme le génie logiciel est l'art de produire des logiciels de qualité, il faut préciser ce que nous entendons par un "logiciel de qualité". Les qualités d'un logiciel concernent à la fois son utilisation (fiabilité, adéquation aux besoins, efficacité, ...) et sa maintenance (flexibilité, portabilité,...). Ces différentes qualités ne sont pas toujours compatibles ni même réalisables et il est nécessaire de trouver des compromis. Dans tous les cas, les objectifs de qualité doivent être définis pour chaque logiciel et la qualité du logiciel doit être contrôlée par rapport à ces objectifs.

Une des premières difficultés concerne la phase de spécification d'un logiciel et plus particulièrement le problème de la vérification des modèles formels par rapport à l'expression informelle des besoins. La difficulté réside dans la différence de langages entre le client (celui qui commande un logiciel selon des besoins plus ou moins précis) et les concepteurs du logiciels (ceux qui conçoivent le logiciel demandé). Le premier s'exprime dans le langage du domaine de l'application du logiciel avec pour support le langage naturel. De fait, les désirs du client sont généralement ambigus et incomplets. Les concepteurs du logiciels utilisent, quant à eux, des méthodes de modélisation comme la modélisation UML [RJB04] ou des méthodes formelles comme la méthode B [Abr96] qui sont généralement mal ou non comprises par le client.

Les activités de vérification et de validation [WF89] ont pour but d'augmenter la sûreté de logiciels. Selon les définitions communément admises, la vérification permet d'établir la correspondance entre un produit logiciel et sa spécification. La validation permet théoriquement de s'assurer que le logiciel accomplit bien la fonction pour laquelle il a été conçu. Dit plus simplement, la validation a pour objectif "le bon" logiciel alors que la vérification a pour objectif le logiciel "bien fait". En réalité, par vérification et validation, on désigne un ensemble de techniques portant sur tout le cycle de développement du logiciel et sur tous ses produits intermédiaires. Ainsi, les revues et les inspections de documents logiciels [1] ainsi que le test font partie de ces techniques.

Les activités de validation et de vérification se déroulent bien souvent conjointement. Deux des principales techniques de vérification et validation sont la preuve et le test de logiciel.

La preuve consiste à démontrer mathématiquement et formellement la correspondance exacte entre un modèle de l'implantation d'une fonction et sa spécification. Il s'agit d'une technique permettant d'avoir un niveau de confiance élevé dans le logiciel mais il s'agit aussi d'une technique coûteuse et incomplète pour les logiciels complexes.

Notons que nous ne nous intéresserons pas à la preuve dans ce manuscrit. Nous nous consacrerons uniquement au test de logiciel.

Le test consiste quant à lui à montrer la conformité d'une implantation à la spécification de la fonction sur un nombre fini d'exécutions (ou de simulation d'exécutions) de la fonction afin d'y trouver un maximum de défauts. Myers [Mye79] définit un défaut de logiciel de la façon suivante : *"A software error is present when the program does not do what its end user reasonability expects to do."*. Il est important de préciser que même si une méthode de test ne détecte pas de défauts dans un logiciel, celui-ci ne peut pas être jugé comme 100% fiable. La citation de Edsger W. Dijkstra, extrait de "Notes On Structured Programming" en 1970, illustre bien ce point : *"Program testing can be used to show the presence of bugs, but never to show their absence !."*. Idéalement, la technique de test exhaustif permet de s'assurer de la correction totale d'un programme et permet également de faire la preuve de correction d'un programme. Cette technique consiste à exécuter un logiciel sur toutes les combinaisons des valeurs en entrée possibles. Par exemple, le test exhaustif d'une simple fonction possédant une unique valeur entière en entrée codée sur 32 bits impose 2^{32} cas de test successifs. On s'aperçoit que cette technique est malheureusement inapplicable en réalité à cause du nombre de tests qu'elle requiert.

Le test de logiciel peut être distingué en trois niveaux de test selon le type de composant logiciel testé. Le premier niveau de test logiciel, le test unitaire, a pour objectif de vérifier que des modules individuels composant un logiciel fonctionnent correctement. Le but du test unitaire est d'isoler chaque module individuel (fonction, méthode, classe, ...) d'un logiciel pour montrer sa correction. Cette étape permet de faciliter le second niveau de test à savoir le test d'intégration. Ce second niveau de test consiste, quant à lui, à tester l'agencement des différents modules d'un système ainsi que la façon dont ils communiquent entre eux [Bei90] pour évaluer la correction de l'ensemble.

Nous précisons que notre travail concerne les langages de programmation impératifs (en particulier le langage C), nous ne traitons donc pas la dimension objet. Ainsi, dans ce manuscrit, par

[1] Les documents logiciels sont tous les documents concernant un logiciel donné à savoir l'expression des besoins en langage naturel, la spécification en langage ou modélisation plus ou moins formel, code source,...)

module individuel nous entendrons une fonction composant le logiciel sous test.

La méthodologie usuelle préconise de tester individuellement chaque fonction d'un logiciel avant de les regrouper pour le test d'intégration afin de pouvoir vérifier leur inter-fonctionnement. En pratique, cela n'est pas aussi simple. Le fait que certaines fonctions d'un logiciel interagissent et/ou utilisent d'autres fonctions d'un même logiciel complexifie le test unitaire. En effet, il n'est pas aisé d'isoler le test unitaire d'une fonction en présence d'appels de fonctions. On est souvent amené à tester d'autres fonctions du fait qu'il existe peu de fonctions réellement indépendantes ou encore à remplacer les fonctions appelées par des modules simulant leurs exécutions. Ainsi, la frontière entre le test unitaire et le test d'intégration est très fine et même si, théoriquement, les deux méthodes sont clairement définies comme différentes, en pratique les deux méthodes sont étroitement liées. Il existe, en pratique, un compromis entre le test unitaire et le test d'intégration. Il s'agit dans un premier temps de définir une hiérarchisation d'appels du logiciel sous forme d'un graphe d'appel à partir de l'étude du langage de programmation utilisé ou de la description de la spécification du logiciel. Deux méthodes de test d'intégration se distinguent particulièrement selon l'ordre d'intégration utilisée pour regrouper les différentes fonctions du logiciel sous test. La méthode "bottom-up" repose sur une agrégation en priorité des fonctions de plus bas niveau dans la hiérarchie d'appels (les fonctions appelées) représentées par les feuilles dans un graphe d'appel. Les fonctions font toutes, au préalable, l'objet de tests unitaires. Ensuite, les différentes fonctions sont intégrées par étapes intermédiaires via une exploration ascendante du graphe d'appel. La méthode "top-down" débute, quant à elle, en intégrant d'abord les fonctions de haut niveau (les fonctions appelantes). La racine du graphe d'appel représente la fonction de plus haut niveau. Il s'agit d'une intégration des fonctions du logiciel correspondant à une exploration descendante du graphe d'appel.

Pour les systèmes reposant sur de multiples interactions et communications entre modules (comme les protocoles de communication, les systèmes de contrôles avioniques,...) des techniques dites de test imbriqué sont appliquées. Elles consistent à tester unitairement un composant donné dans un environnement précis c'est-à-dire au sein du groupement composé des autres composants du système. Ces techniques se justifient par la complexité du système sous test ne permettant pas de tester unitairement et indépendamment les composants du système en totalité. Le but du test imbriqué est de vérifier qu'un composant possède un comportement à l'exécution conforme à celui de sa spécification lorsqu'il interagit avec les autres composants.

En quelques mots, le test d'intégration correspond à une technique de test unitaire étendue sur les agrégations successives des fonctions d'un logiciel sous test. Les fonctions non encore intégrées sont simulées et remplacées par des bouchons pour les fonctions de bas niveau et par des lanceurs pour des fonctions de haut niveau [Mye79] simulant le comportement des fonctions remplacées. Parallèlement, le test unitaire d'une fonction utilise également ces mêmes bouchons pour remplacer les appels de fonctions contenus dans la fonction testée individuellement. Du fait de cette ambiguïté entre le test unitaire et le test d'intégration, la littérature sur le test d'intégration reste encore assez faible aujourd'hui. Cependant, il existe des techniques propres au test d'intégration orientées vers l'évaluation des connexions entre les fonctions d'un logiciel comme [OHK93]. Lors du test imbriqué, le lien entre le composant sous test et les autres composants du système est évident du fait que le test imbriqué est destiné aux systèmes composés de modules fortement dépendants les uns des autres. Nous verrons un panorama plus détaillé des techniques de test d'intégration et des techniques de test imbriqué dans le chapitre 10.

Revenons maintenant sur la problématique du test unitaire. Trois classes de méthodes de test se distinguent par les critères de sélection des cas de test utilisés. Commençons par les techniques de test aléatoire consistant à injecter « au hasard » différentes valeurs aux variables d'entrée de la fonction sous test. Le composant est exécuté avec des données choisies aléatoirement dans le domaine de définition de la fonction. En pratique, plus le nombre de cas de test est grand, plus le pourcentage d'objets testés est élevé [GDGM01]. La difficulté de ces techniques vient du fait qu'elles ne garantissent pas d'atteindre tous les comportements de la fonction et en particulier les comportements dont la probabilité d'exécution est faible (comportement associé à un domaine en

entrée restreint de la fonction).

D'autre part, les techniques de test fonctionnel (ou méthodes boîte noire) déterminent les différentes données de test en s'appuyant sur les documents de spécification de la fonction. Le plus souvent, ces techniques reposent sur un partitionnement du domaine d'entrée correspondant aux différentes fonctionnalités de la fonction exprimées dans la spécification. Elles permettent de choisir le niveau de détails des tests par le choix du niveau d'abstraction de la spécification utilisée.

Enfin, les méthodes de test structurel (ou méthodes boîte blanche) sont les méthodes les plus communément admises. Elles consistent à déterminer les différentes données de test par analyse du code source selon une étude flot de données[2] ou selon une étude flot de contrôle[3] de la fonction sous test. Les critères de flot de données sont basés sur la couverture des chemins reliants les définitions[4] d'une variable à leur utilisation[5] [RW85]. Le but est de s'assurer que la totalité des différentes exécutions possibles (ou un nombre maximal) du logiciel a bien été explorée.

A la fin de chaque cas de test, il faut pouvoir émettre un verdict de test c'est-à-dire définir si les sorties obtenues à l'exécution sont conformes à celles attendues telles qu'elles sont définies dans la spécification. Il s'agit du problème de l'oracle. Pour une méthode structurelle, s'appuyant sur le code source de la fonction, la spécification de la fonction est nécessaire pour prévoir les valeurs en sortie attendues. La spécification de la fonction est utilisée pour comparer les valeurs en sortie obtenues lors des cas de test avec celles attendues ou encore pour vérifier que les valeurs en sortie obtenues à l'exécution respectent bien les propriétés spécifiées. Pour une méthode fonctionnelle, la mise en place d'un verdict de test est facilitée du fait que les différentes données de test sont déterminées par l'étude de la spécification de la fonction sous test. L'utilisation d'assertions dans le code source de la fonction sous test en programmation défensive ou dans l'interface sous forme de contrats [Mey92] introduit par le langage Eiffel[6] de Meyer [Mey88], joue le rôle d'oracle embarqué. Meyer est le premier à avoir utilisé les spécifications exprimées sous forme de couples pre/post[7] d'une fonction sous test comme oracle embarqué exprimé par des assertions dans le code. Les assertions contiennent des propriétés de la fonction à vérifier à un instant précis de l'exécution [TBJ06],[CL01]. Une autre approche de mise en place d'oracle consiste à calculer les sorties attendues de la fonction en se basant sur la spécification et de comparer le résultat aux sorties réelles obtenues lors des test [PP94], [GA95], [RAO92]. Notons que le problème de l'oracle reste encore trop peu abordé dans la littérature à l'exception des approches formelles.

L'automatisation des cas de test permet de gagner en temps et par conséquent en coût de la phase de test d'une fonction. Dans la pratique industrielle, l'automatisation du test est trop souvent limitée à l'exécution des tests. C'est l'utilisateur qui définit les cas à tester et les données de test (entrées et sorties attendues), éventuellement à l'aide d'outils de préparation de "scripts de test". La génération automatique des données de test et d'un oracle permet une sélection des cas de test basée sur des critères justifiés par des hypothèses explicites. L'automatisation complète d'une méthode de test permet la réduction des coûts et des délais de vérification des produits logiciels tout en améliorant la qualité du logiciel. Les techniques fonctionnelles sont privilégiées pour la génération automatique de cas de test comme pour les outils BZ-TT[ABC+02], GATEL [MA00], Casting [ABM97], LUTESS [BORZ99], AUTOFOCUS[PO01], ... La difficulté principale de l'automatisation du test de logiciel concerne en particulier l'automatisation de l'oracle. Actuellement, l'utilisation de spécifications formelles reste le moyen reconnu comme le plus adapté pour l'automatisation d'oracles fiables [GA95]. L'utilisation des méthodes formelles est recommandée pour le développement de logiciels critiques. Leur sémantique bien définie permet une spécification précise du service à délivrer. Elles sont fondées sur des bases mathématiques permettant éventuellement d'effectuer des preuves pour démontrer certaines propriétés de la spécification ou pour vérifier de

[2] Une étude flot de donnée se base sur les données manipulées ou utiles et sur leurs définitions et leurs utilisations.

[3] Une étude flot de contrôle se base sur les enchaînements possibles des différentes instructions du composant sous test.

[4] La définition d'une variable correspond à une modification de la valeur de la case mémoire associée

[5] L'utilisation d'une variable correspond à l'accès en lecture de la case mémoire associée.

[6] Le langage Eiffel est un langage de programmation par contrat orienté objet implanté par Bertrand Meyer.

[7] Les couples pre/post sont des propriétés sur les variables de la fonction sous test à vérifier en différents instants de l'exécution (avant l'exécution pour les preconditions, après pour les postconditions, à chaque instant pour les invariants, ...

manière formelle les phases de conception et d'implantation.

L'arrêt du test survient quand le critère d'arrêt (plus ou moins formel) est satisfait. Pour le test unitaire, l'arrêt du test est directement déterminé par les critères de test sélectionnés. Idéalement, le test s'arrête quand :

- tous les objets cibles du graphe de contrôle ou
- tous les objets du graphe de flot de données de la fonction ou
- tous les domaines d'entrée de la fonction associés à une fonctionnalité spécifiée

ont été couverts. En pratique, pour des critères de test exigeants comme les critères structurels (en particulier le critère *tous-les-chemins*), le test peut être arrêté lorsqu'un pourcentage fixé (taux de couverture) des objets du graphe de contrôle ou du graphe de flot de données a été couvert. Les méthodes orientées flot de données comme [KL85] sont reconnues comme efficaces dans différentes études empiriques [Won93], [HFGO94]. L'automatisation de ces méthodes est complexifiée par l'étude précise des définitions et utilisations de chaque variable de la fonction sous test. Le plus rigoureux critère structurel orienté flot de données c'est-à-dire le critère *"all-du-path"*[8] est reconnu comme moins rigoureux que le critère orienté flot de contrôle *tous-les-chemins*. Nous avons choisi d'adopter ce critère d'une part pour sa rigueur mais aussi du fait que notre stratégie de sélection des cas de test permet de limiter les difficultés propres à ce critère à savoir la détection des chemins infaisables et l'explosion combinatoire du nombre des chemins tout en limitant le coût du calcul de la détermination des données de test.

La troisième technique de test structurel est le test mutationnel. Il s'agit de générer des mutants du code source de la fonction sous test en modifiant une seule de ses instructions. Le but est de générer une suite de données de test permettant de tuer ces mutants ou un nombre maximal (c'est-à-dire de détecter les anomalies injectées à la fonction). Le test mutationnel reste plus une approche pour évaluer la qualité d'une suite de données de test. L'étude empirique [ABL05] montre l'efficacité de détection des fautes des jeux de test définis lors de techniques de test mutationnel par rapport à des suites de test générées par d'autres approches souvent plus coûteuses. Les méthodes de test mutationnel [NFTJ06], [BDL06], [OAL06] se heurtent cependant à la difficulté de mise en œuvre de cette approche en particulier en présence de mutants équivalents (les fautes générées peuvent ne pas avoir d'influence sur l'exécution de la fonction). Cela impose une inspection supplémentaire empêchant l'automatisation de ces méthodes.

Nous nous intéresserons donc uniquement aux techniques de test structurel basées sur l'étude du flot de contrôle des fonctions sous test et en particulier sur l'application du critère tous-les-chemins.

Nous sommes forcés de constater que, malgré la rigueur des techniques structurelles, celles-ci restent peu appliquées en pratique à cause de la complexité du passage à l'échelle à des fonctions réalistes complexes (présence de boucles, appel à d'autres fonctions, ...) et à cause de la difficulté de mise en œuvre des jeux de test satisfaisant les critères considérés (présence de chemins inexécutables, difficulté de la mise en place d'un oracle, ...).

Nous proposons ici une nouvelle approche pour le test unitaire structurel visant d'une part à faciliter la détermination et la génération des cas de test et d'autre part à automatiser au maximum la génération des tests.

1.2 Sujet

Les méthodes de test structurel sont largement reconnues pour leur rigueur. Le test structurel est tout particulièrement requis pour les logiciels critiques qui doivent satisfaire certains critères imposés par les organismes normatifs (ISO, AFNOR, ...). Elles représentent un coût non négligeable dans le développement des logiciels lorsque les exigences de sécurité sont importantes. Ainsi, pour le test de logiciels critiques, il est régulièrement imposé d'atteindre une couverture complète de certains objets du graphe de flot de contrôle (CFG) comme les nœuds, les arcs et les chemins représentant respectivement les instructions, les branchements ou les chemins d'exécution

[8] Ce critère correspond à couvrir pour chaque définition d'une variable tous les chemins contenant une utilisation de cette même variable.

du code source [GBR00], [GDGM01], [Meu01], [WMM04b]. Le but est de s'assurer que la totalité des différents objets issus du CFG choisis comme cibles du critère de test a bien été explorée avec les exécutions sélectionnées lors des différents cas de test effectués.

Dans la classe des critères définis à partir du CFG, le critère *tous-les-chemins* est le critère le plus complet, exigeant et demandant le plus grand nombre de cas de tests. Pour les projets logiciels industriels concernant des logiciels de faible ou moyenne complexité, l'application du critère *tous-les-chemins* peut être imposée. Cependant, le plus souvent, la totalité des chemins exécutables n'est pas couverte et on se contente d'atteindre un taux de couverture le plus élevé possible. Ainsi, pour des logiciels plus complexes c'est-à-dire réalistes, l'application du critère *tous-les-branchements* (couverture de toutes les branches du graphe de contrôle de la fonction sous test) est souvent requise. Un autre critère structurel orienté flot de contrôle souvent imposé est le critère *MC/DC* correspondant à une variante plus rigoureuse du critère *tous-les-branchements* (couverture individuelle de tous les sous-conditions composant une instruction conditionnelle). Pour chaque fonction possédant n structures conditionnelles successives, il peut y avoir 2^n chemins différents d'exécutions. Par conséquent, la présence de structures répétitives (ou boucles) peut résulter à une infinité de chemins différents. Il est alors parfois nécessaire de limiter le nombre de passages dans ces structures répétitives : le critère de test *tous-les-chemins* est restreint alors au critère des *k-chemins*. Il s'agit de couvrir tous les chemins d'exécution ne contenant pas plus de k itérations par structure répétitive.

Notons que plusieurs chemins peuvent, de plus, être infaisables c'est-à-dire qu'aucune entrée de la fonction n'entraîne l'exécution de ces chemins. Cette difficulté est non négligeable : c'est un problème indécidable dans le cas général et NP-complet pour des variables prenant leurs valeurs dans des domaines finis.

Toutes les difficultés associées au test structurel complexifient ou empêchent l'automatisation des techniques de test associées.

Le test structurel peut se décomposer en deux étapes :

- l'identification de chemin(s) dont l'exécution est nécessaire pour satisfaire un critère de test,
- la génération d'un cas de test (i.e. la détermination d'un ensemble de valeurs pour les variables d'entrée) garantissant que le chemin (ou un des chemins) sélectionné(s) sera effectivement exécuté.

Les difficultés liées à l'analyse du code source sont nombreuses. D'une part, les langages de programmation ne possèdent pas en général de forme canonique, il s'agit le plus souvent d'uniformiser l'implantation d'une fonction par un prétraitement du code visant à atteindre une forme canonique facilitant l'analyse. D'autre part, pour chaque chemin d'exécution suivi, il s'agit de construire le prédicat de chemin associé correspondant à la conjonction des contraintes sur les valeurs initiales en entrée entraînant l'exécution de ce chemin. Le calcul de ce prédicat de chemin demande un effort non négligeable lors de l'analyse du code source des fonctions (expressions à effets de bord, instructions de modification de flot de contrôle, présence d'alias, gestion des types flottants entre autres). Les techniques de test structurel possèdent de ce fait certaines restrictions quant à l'ensemble du langage de programmation traité.

Une autre des difficultés majeures liées aux méthodes structurelles concerne l'explosion combinatoire du nombre de chemins en présence de structures répétitives comme nous l'avons précisé précédemment mais aussi en présence d'appels de fonctions. Quand le critère structurel choisi est *tous-les-chemins*, l'explosion combinatoire concerne, par voie de conséquence, le nombre de cas de test à effectuer. Pour la gestion des appels de fonctions, deux méthodes sont couramment utilisées. La méthode "inlining" consiste à déplier le code source des fonctions appelées dans le code source de la fonction appelante pour y étendre le critère de test. Cette méthode amplifie le problème de l'explosion combinatoire du nombre des chemins en ajoutant la combinatoire du nombre des chemins des fonctions appelées à la combinatoire du nombre des chemins de la fonction sous test. Une autre méthode consiste à substituer les appels de fonctions par des modules simulant le comportement de la fonction appelée par l'étude de ses spécifications (bouchons fonctionnels) ou retournant des valeurs données en vue de la couverture de la fonction sous test (bouchons structurels) [Mye79] sans prendre en compte le comportement réel des fonctions appelées. Les bouchons fonctionnels

retournent une sortie pour une entrée donnée de la fonction conformément à sa spécification : la relation entrées/sorties est ainsi vérifiée par construction. Pour un contexte d'appel et un domaine fonctionnel donné d'une fonction appelée, différents chemins en sortie peuvent être exécutés dans la fonction sous test. Il faut alors pouvoir orienter l'exécution des bouchons fonctionnels pour pouvoir garantir le maintien de la couverture de la fonction sous test. Or, les bouchons fonctionnels sont assimilables à des composants boîtes noires. Les bouchons structurels, quant à eux, peuvent amener à couvrir des chemins infaisables en réalité du fait que le comportement réel des fonctions appelées est ignoré.

Ce manque d'outillage quant à la gestion des appels de fonction empêche le passage à l'échelle du test unitaire de fonctions réalistes.

Nous adressons cette difficulté de mise en place d'une méthode automatique de test structurel. Nous traitons en particulier du passage à l'échelle des techniques structurelles de test pour le test de fonctions dont le code fait appel à d'autres fonctions et de l'automatisation de ces techniques. Nous proposons ici une nouvelle méthode de génération automatique de cas de test structurels [WMM04b] ainsi qu'une nouvelle gestion des appels de fonction afin d'assurer la couverture structurelle d'une fonction sous test [Mou07] . La méthode automatique de test structurel proposée répond au critère structurel *tous-les-chemins* ou sa variante, le critère des *k-chemins*. Cette méthode permet d'atteindre une couverture totale des k-chemins faisables de la fonction sous test en facilitant le problème de la détection des chemins infaisables pour un coût limité lors de la détermination des données de test successives.

En ce qui concerne la gestion des appels de fonction, nous avons choisi de combiner les aspects fonctionnels et structurels de test pour profiter de l'ensemble de leurs avantages.

Nous sommes partis de l'idée d'exploiter les spécifications des fonctions pour le test d'une fonction dont le code fait appel à d'autres fonctions. Nous proposons une nouvelle modélisation des fonctions sous test avec instructions d'appel reposant à la fois sur les informations structurelles de la fonction sous test et sur les informations fonctionnelles issues de la spécification des fonctions appelées. Nous proposons également deux nouveaux critères de test structurels permettant de garantir le maintien de la couverture de la fonction sous test tout en limitant l'explosion combinatoire du nombre de chemins provoquée par les appels de fonctions. Notre objectif est de pouvoir maintenir la couverture structurelle de la fonction sous test tout en limitant l'exploration des fonctions appelées afin de limiter le problème de l'explosion combinatoire du nombre des chemins.

1.3 Plan

La première partie de ce manuscrit présente le contexte général de la thèse.

Le chapitre 2 décrit et justifie le sous-ensemble du langage C considéré par notre approche. La quasi-totalité du langage C ANSI est traitée. Les codes sources des fonctions sont soumis à un prétraitement de façon à ce que les codes prétraités appartiennent à un sous-ensemble précis du langage C ANSI. Nous décrivons donc ce sous-ensemble du langage C qui correspond au sous-ensemble réellement manipulé lors de la sélection et de la génération des différents cas de test.

Le chapitre 3 caractérise le langage de spécification utilisé pour modéliser les fonctions appelées. Ce langage de spécification correspond à un langage du premier ordre sur domaines finis. Nous présentons également le format choisi pour exprimer les spécifications des fonctions à savoir une axiomatisation sous forme de couples pre/post.

Le chapitre 4 présente la représentation courante du code source des fonctions sous test à savoir le graphe de contrôle d'une fonction (CFG). Ce chapitre introduit également la notion de chemin d'un CFG et la notion associée de prédicat de chemin. Nous insistons sur la notion de CFG dans la mesure où nous proposons une nouvelle modélisation des fonctions sous test avec appels s'apparentant à la représentation classique d'un code source sous forme de CFG.

Le dernier chapitre de cette partie, le chapitre 5, est consacré à la présentation du test de logiciel et des techniques les plus couramment utilisées. Les lecteurs familiers au test de logiciel pourront ignorer ce chapitre s'ils le désirent mais nous insistons sur le fait que ce

chapitre permet également de fixer clairement la terminologie utilisée tout au long de ce manuscrit.

La seconde partie de ce document est consacrée au problème de la génération automatique de cas de test structurel.

Le chapitre 6 définit la génération automatique de cas de test structurel, sa problématique ainsi que ses principales difficultés. Un rapide panorama de techniques consacrées à la génération automatique de cas de test structurel y est également dressé.

Le chapitre 7 présente en détails notre méthode automatique de génération de cas de test structurel nommée PathCrawler. Cette méthode respecte le critère de test structurel le plus rigoureux : le critère *tous-les-chemins* ou sa variante, le critère des *k-chemins* en présence de structures répétitives à nombre élevé d'itérations. Cette méthode adaptative réutilise le prédicat de chemin du cas de test courant pour déterminer les données du prochain cas de test. Les informations extraites du prédicat de chemin précédent permet de faciliter la détection des chemins infaisables tout en limitant le coût de la détermination du prochain cas de test.

La chapitre 8 illustre plus en détails notre méthode de génération de cas de test structurels par son application sur un exemple. Nous discutons également des difficultés de passage à l'échelle de cette méthode en présence d'instructions d'appel.

Enfin, la troisième partie est, quant à elle, consacrée au traitement particulier des appels de fonctions.

Dans le chapitre 9, nous faisons tout d'abord une rapide présentation de notre gestion des appels de fonctions et nous définissons la problématique associée.

Le chapitre 10 présente différentes techniques de test chargées d'évaluer un ensemble de composants entre eux ou un composant dans un environnement donné (respectivement le test d'intégration et le test imbriqué). Nous présentons également plus en détails les techniques d'"inlining" et d'utilisation de bouchons, techniques le plus souvent utilisées pour la gestion des appels de fonction. Nous pouvons ainsi nous situer de façon plus précise par rapport aux approches existantes.

Le chapitre 11 décrit en détails la mise en œuvre de notre approche de gestion des appels. Nous avons choisi d'abstraire les fonctions appelées par l'étude de leurs spécifications. Nous construisons ainsi une représentation fonctionnelle des fonctions appelées désignée comme graphe abstrait. Cette représentation fonctionnelle des fonctions appelées se substitue aux instructions d'appel dans le code source de la fonction sous test lors de la construction de son graphe de contrôle. Nous obtenons une nouvelle modélisation des fonctions sous test avec instructions d'appel sous forme d'un graphe mixte comportant à la fois des informations structurelles de la fonction sous test et fonctionnelles des fonctions appelées. Nous définissons deux nouveaux critères de test dédiés à cette représentation garantissant le maintien de la couverture structurelle de la fonction sous test et la limitation de l'exploration des fonctions appelées.

Le chapitre 12 décrit l'application de notre modélisation des fonctions sous test sous forme de graphe mixte dans le cadre de la méthode de génération de cas de test PathCrawler. Nous définissons ensuite les deux nouveaux critères de test dédiés à cette représentation sous forme de graphe mixte.

Dans le chapitre 13, nous validons notre approche par son application à deux exemples. Ces exemples de fonctions sous test avec instructions d'appels sont soumis, dans un premier temps, aux deux traitements classiques d'"inlining" et d'utilisation de bouchons. Ensuite, ces exemples sont soumis à l'application de nos deux critères de test appliqués au graphe mixte des fonctions sous test. Les résultats obtenus permettent d'illustrer les gains et avantages de notre approche par rapport aux techniques plus classiques peu adaptées.

Enfin, le dernier chapitre, la conclusion, permettra de dresser un bilan précis de notre travail. Nous discutons du respect des objectifs initiaux et des améliorations envisagées de la méthode. Nous expliquons également notre contribution en donnant quelques-unes des orientations envisagées pour la suite.

Première partie

Contexte du mémoire

Cette partie a pour objectif de définir toutes les notions nécessaires pour la suite de ce document. Nous avons en effet besoin d'établir clairement certaines notions pour aborder les deux thèmes principaux de ce manuscrit à savoir le test unitaire et le test d'intégration.

Dans un premier chapitre, nous donnerons les prérequis nécessaires pour la lecture de ce manuscrit et nous nous attarderons sur certains points du langage étudié à savoir le langage C.

Ensuite, nous consacrerons un chapitre à la spécification des fonctions C à l'aide de formules de types pre/post.

Dans le chapitre suivant, nous présenterons la notion de graphe de flot de contrôle comme la représentation interne d'une fonction C utilisée communément par les méthodes de test structurel.

Enfin, un dernier chapitre dressera un panorama des principales techniques de test de logiciel et contiendra un rappel de la terminologie utilisée.

Chapitre 2

Langage C

La méthode présentée dans ce document s'adresse aux langages impératifs séquentiels et en particulier au langage C, langage pour lequel nous avons implanté un prototype de notre stratégie.

Nous supposons le lecteur familier avec le langage C. Cependant en cas de besoin, le lecteur pourra se référer aux multiples ouvrages dédiés à ce langage comme [Ame86].

Ce qui nous intéresse dans ce document par rapport au langage C est d'une part la caractérisation de la notion de flot de contrôle d'un programme ainsi que l'explicitation des notions de variables d'entrée et de sortie d'une fonction C.

Nous nous limitons dans ce chapitre à un sous-ensemble du langage C correspondant au sous-ensemble résultant du prétraitement des fonctions (cf. section 7.2) sous test opéré en amont de la génération des cas de test dans notre méthode de test structurel PathCrawler. Ce prétraitement est basé sur l'utilisation de l'analyseur syntaxique CIL [Lan06]. Nous insistons sur le fait que nous traitons la quasi totalité du langage C ANSI à quelques restrictions près données dans la section 7.2.1 et sur le fait que le prétraitement effectué en amont de la génération des cas de test par la méthode PathCrawler simplifie et transforme certaines structures du langage sous formes canoniques. Nous verrons également plus tard que le prétraitement modifie également le flot de contrôle de la fonction.

L'annexe A présente d'autres notions du langage C n'appartenant pas au sous-ensemble résultant du prétraitement mais qui seront utiles pour la suite.

2.1 Types de base

En ce qui concerne le typage des variables, il existe deux familles principales. D'une part, nous avons les types simples appelés types de bases et d'autre part, nous avons les types complexes ou types structurés résultant d'une construction de types dont la base repose sur un ou plusieurs types de bases (comme par exemple une variable de type tableau contenant n variables d'un type de base). Nous verrons dans la section suivante que les types de bases sont suffisants pour notre méthode de génération de cas de test dans la mesure où les types structures sont totalement décomposés.

Chaque type de base est associé à un intervalle de valeurs muni d'un ordre total dont la taille est directement dépendante de la taille en octets associée. Le tableau 2.1 nous donne ces intervalles de valeurs pour chaque type C de base selon la taille en octets associée.

Nous réutiliserons ces mêmes intervalles par la suite comme les domaines associés aux sortes de base de notre langage de spécification (cf. chapitre 3).

Type de base C	Taille (en octets)	Intervalle associé
int	4	$[-2^{31}; 2^{31}-1]$
short int	2	$[-2^{15}; 2^{15}-1]$
long int	4	$[-2^{31}; 2^{31}-1]$
unsigned short int	2	$[0; 2^{16}-1]$
unsigned long int	4	$[0; 2^{32}-1]$
signed char	1	$[-2^{7}; 2^{7}-1]$
unsigned char	1	$[0; 2^{8}-1]$
float	4	$[-3.4^{38}; 3.4^{38}]$(7 chiffres significatifs)
double (long float)	8	$[-1.7^{308}; 1.7^{308}]$ (15 chiffres significatifs)
long double (non standard)	10	$-3.4^{4932}1.1^{4932}$ (quadruple précision)

TAB. 2.1 – Intervalles des types de base et taille associée en octets

2.2 Notion de variables

Nous considérons comme des variables toute variable C déclarée de type de base ainsi que chaque élément de type de base des variables structurées. Si un élément de variable de type structuré est également de type structuré, nous décomposons cet élément jusqu'à en déterminer les éléments de type de base associés.

Illustration 1
La déclaration de la variable C suivante : `"int tab[2];"` *correspondant à un tableau à 2 éléments de type entier induit selon notre définition deux variables* `t[0]` *et* `t[1]` *de type* `int`.

Les variables dans le sens où nous l'entendons sont donc toutes d'un type de base.

Introduisons dès à présent un concept primordial pour la suite : la notion de flot de contrôle.

Le **flot de contrôle** réfère à l'ordre dans lequel les instructions individuelles d'un programme impératif sont exécutées. Le flot de contrôle est influencé par les différentes structures notamment conditionnelles ou répétitives du langage rencontrées le long d'une exécution donnée et dépend donc de la satisfaction des conditions associées aux structures du langage et ainsi des valeurs en entrée injectées au programme (nous reviendrons sur ce point dans la section 2.6.5). L'évaluation des structures rencontrées en fonction des valeurs en entrée d'une fonction modifie donc les séquences d'instructions exécutées. C'est cette variabilité des séquences d'instructions qui reflète la notion de flot de contrôle.

2.3 Structures du langage

Le langage C admet des expressions construites à partir des différents opérateurs du langage. Ces expressions seront notées de façon générique `exp`.

Il existe des instructions élémentaires (calculs sur des variables) et des instructions conditionnelles ou répétitives. Nous ne nous attarderons pas ici sur la sémantique des différentes opérations élémentaires du langage C (comme la division, l'addition etc.). En revanche, nous allons parler de l'utilisation de structures conditionnelles et répétitives utilisées dans le langage après avoir présenté l'instruction élémentaire du langage à savoir l'affectation.

2.3.1 Instruction élémentaire : l'affectation

En langage C, l'opérateur `=` est un opérateur que l'on peut interpréter comme l'action de donner la valeur du membre droit au membre gauche.

Pour une affectation de la forme :

```
"id = exp ;"
```

avec `id` l'identifiant d'une variable non constante déclaré avec un certain type et `exp` une expression du même type alors l'évaluation de cette affectation, forcée par `" ;"`, a pour effet d'évaluer l'expression `exp` et de remplacer dans la mémoire le contenu de la case dont l'adresse correspond à l'identifiant `id` par la valeur de `exp`. On dit alors communément que la valeur des variables de `exp` sont utilisées pour définir la valeur de `id`.

La notion de **définition d'une variable** désigne une modification de la valeur de celle-ci et la notion d'**utilisation** d'une variable correspond à son évaluation c'est-à-dire à la lecture de sa valeur.

Illustration 2
L'exécution de l'affectation `"tab[i+2]=3*x ;"` signifie que la valeur correspondante à la variable `tab[i+2]` prend la valeur de l'expression `3*x`.

L'affectation est une instruction qui modifie l'état de la mémoire du programme.

Les autres structures du langage C sur lesquelles nous allons nous attarder reposent sur l'évaluation et la vérification d'une expression C appelée condition. Ce point nous amène donc à préciser tout d'abord la notion de vérification d'une condition en langage C.

2.3.2 Évaluation d'une expression et vérification d'une condition

Il n'existe pas dans le langage C de type de base composé de 2 constantes bien identifiées. Néanmoins, il est possible de distinguer les expressions en les évaluant afin de les classer respectivement en des expressions considérées comme vraies et fausses.

Nous avons, d'une part, le test de validité d'une expression ou d'une donnée C : une expression est non valide si sa valeur vaut zéro et valide dans le cas contraire.
D'autre part, lorsque le résultat d'une expression logique i.e. une expression contenant des opérateurs logiques est une valeur différente de zéro, cela dénote en langage C la valeur vraie alors que lorsque la valeur résultante vaut zéro, il s'agit d'une expression considérée comme fausse.

Illustration 3
Le résultat de l'expression :

- *`!expr1` est vérifiée si `expr1` est évaluée à 0 (`!` est l'opérateur C de la négation logique) ;*
- *`expr1&&expr2` est vérifiée si et seulement si les deux expressions `expr1` et `expr2` sont vérifiées i.e. ne sont pas évaluées à 0 (`&&` est le "et-logique" du langage C) ;*
- *`expr1||expr2` est vérifiée si et seulement si l'une au moins des expressions `expr1, expr2` n'est pas évaluée à 0 (`||` est le "ou-logique" du langage C).*

2.3.3 Structures conditionnelles

Une structure conditionnelle permet à un programme de décider ou non d'exécuter une séquence d'instructions en se basant sur la valeur d'une expression appelée communément la condition de la structure conditionnelle.

Du fait que la valeur d'une expression peut varier d'une exécution à une autre, cela permet à la fonction de réagir dynamiquement aux différentes valeurs de l'expression évaluée.

Une **structure conditionnelle** est donc une instruction C qui entraîne une modification du flot de contrôle d'une fonction selon qu'une condition soit vérifiée ou non.

Les deux structures conditionnelles conservées après prétraitement du code source des fonctions sous test sont les structures conditionnelles IfThenElse et IfThen.

Structure IfThenElse

La syntaxe de la première forme d'instruction conditionnelle, la structure conditionnelle IfThenElse, est présentée dans la figure 2.1 dans laquelle `exp, bthen, belse, b` représentent respectivement une expression du langage C et trois blocs d'instructions[1].

```
1   if  (exp)
2    bthen;
3   else
4    belse;
5   b;
```

FIG. 2.1 – Syntaxe C de la structure IfThenElse

La sémantique de la structure IfThenElse consiste à exécuter le bloc d'instructions `bthen` si l'expression `exp` est vérifiée et d'exécuter le bloc d'instructions `belse` dans le cas contraire. L'expression `exp` représente ici soit une variable de type entier soit une expression booléenne sur des variables. `exp` est évaluée selon la notion de validité expliquée dans la section 2.3.2. Deux flots de contrôle différents sont créés selon que l'évaluation de l'expression `exp` est vérifiée ou non. Notons que le bloc d'instructions `b` est quant à lui toujours exécuté après la conditionnelle.

Structure IfThen

La syntaxe de la deuxième forme de structure conditionnelle IfThen est présentée dans la figure 2.2.

```
1   if  (exp)
2    bthen;
3   b;
```

FIG. 2.2 – Syntaxe C de la structure IfThen

La sémantique de la structure IfThen est similaire à celle de la structure conditionnelle précédente IfThenElse à la différence près que si l'expression `exp` n'est pas vérifiée aucune instruction spécifique est exécutée. Sinon, de façon similaire à la structure IfThenElse, le bloc d'instructions `b` est toujours exécuté après la structure conditionnelle. Nous avons toujours la création de deux flots de contrôle différents.

2.3.4 Structure répétitive While

Une **structure répétitive** est une structure qui permet de répéter un certain nombre de fois un bloc d'instructions et ce, jusqu'à la vérification d'une expression de la structure entraînant la sortie de la structure, expression communément appelée condition de sortie de boucle.

[1] Un bloc d'instructions en langage C désigne soit une instruction seule soit un groupement d'instructions délimités par des accolades.

Le langage C contient, outre la structure répétitive While, deux autres types de structures répétitives qui sont expliquées dans l'annexe A.2.

Lors du prétraitement des fonctions sous test (cf. section 7.2), les différentes structures répétitives du langage C sont toutes transformées sous une forme canonique correspondant à une structure répétitive While.

La structure répétitive While du langage C consiste à exécuter plusieurs fois successives un même bloc d'instructions. La figure 2.3 présente sa syntaxe.

```
1   while  (exp)
2     b1;
3   b;
```

FIG. 2.3 – Syntaxe C de la structure While

En ce qui concerne son fonctionnement, le bloc d'instructions `b1` est exécuté tant que l'expression `exp` est vérifiée. En particulier, si l'expression `exp` est toujours vérifiée, l'instruction While ne termine jamais, on parle alors de **bouclage à l'infini**.

Le bloc d'instructions `b` sera exécuté en sortie de la structure While si celle-ci termine.

2.4 Autres instructions de modification de flot de contrôle

L'instruction `break` est utilisée dans les structures répétitives et conditionnelles. Dans une structure répétitive, cette instruction provoque la sortie immédiate de la structure sans tenir compte des conditions de sortie de la structure.

Dans une structure répétitive, l'instruction `continue` produit l'abandon de l'itération courante et, si le condition de sortie l'autorise, le démarrage de l'itération suivante.

L'instruction `goto` transfère directement le contrôle du programme à une instruction étiquetée. Cette instruction permet de continuer l'exécution du programme à un autre endroit, dans la même fonction. On l'utilise de la manière suivante : `"goto label ;"` où `label` est un identificateur quelconque. Cet identificateur devra être défini quelque part dans la même fonction, avec la syntaxe suivante : `"label :/*instructions*/"`. Ce label peut être mis à n'importe quel endroit dans la fonction, y compris dans un autre bloc que là où sont utilisées la ou les instructions `goto` pointant vers ce label. Les labels doivent être uniques au sein d'une même fonction, mais on peut les réutiliser dans une autre fonction. Notons que, actuellement, l'outil d'analyse statique CIL [Lan06] ne permet pas de prendre en compte les gotos[2] arrières (dont le label précède l'instruction `goto` associée dans le code).

2.5 Catégories de variables C

En langage C, on distingue différentes catégories de variables selon leur portée c'est-à-dire selon leur durée de vie dans l'implantation associée. Nous avons les variables dites globales qui ont une portée qui s'étend à tout le programme dans lequel elles sont déclarées et les variables dites locales qui ont une portée qui se limite au bloc dans lequel elles sont déclarées.

Nous allons expliciter ces points plus en détail.

[2] Instruction provoquant un saut vers une autre instruction dans l'implantation d'une fonction

2.5.1 Variables locales

Dans toute fonction du langage C, on peut déclarer localement (i.e dans le corps de cette fonction) des variables dites locales.

Les variables locales ont une portée limitée. Lors du prétraitement du code source des fonctions, toutes les déclarations des variables locales sont remontées au début du bloc principal de la fonction qui les contient.

De façon générale, après le prétraitement, toutes les déclarations des variables locales d'une fonction respectent le schéma suivant :

```
int fonc (...)
{
/*déclarations des variables locales de la fonction */
/*corps de la fonction*/
}
```

Les variables locales sont donc toutes déclarées au début de la fonction i.e. au début du bloc d'instructions principal de la fonction et par conséquent, les variables locales sont accessibles de l'instant où la variable est déclarée à la fin de la fonction.

En langage C, un **conflit de variables** est caractérisé dans un code source par le fait que deux variables (pouvant être de types différents) aient le même nom en n'étant pas déclarées dans un même bloc d'instructions. La dernière variable déclarée cache tout au long de sa durée de vie la variable plus ancienne de même nom.

Illustration 4
La figure 2.4 est une illustration de code source contenant un conflit de variables. En effet, aux lignes 4 et 11, deux variables `i` *sont déclarées. Notons au passage que ces deux variables sont de types différents.*

Lors du prétraitement, outre la remontée des déclarations des variables locales au début du bloc principal de la fonction, les variables locales sont renommées de façon à éviter tout conflit de variables entre les variables locales d'une même fonction et entre les variables locales d'une fonction et les variables globales d'un programme contenant cette fonction.

La portée des variables locales après prétraitement ne peut donc pas être limitée par un conflit de variables.

2.5.2 Variables globales

Les **variables globales** sont les variables déclarées hors de toute fonction et par conséquent hors de tout bloc d'instructions. Leur durée de vie est celle du programme associé. Les conflits de variables sont également écartés lors du prétraitement pour les variables globales.

La déclaration d'une variable globale de type C `int` que nous nommerons `varGlob` suit le schéma suivant :

```
...
int varGlob;
int fonction(...)
{
/* corps de la fonction */
}
```

```
1  int pbvar(void)
2  {
3    int a;
4    int i;           /*début de vie du i entier */
5    char c;
6    a=1;
7    int j;
8    c="c";
9    for(i=0;i<3;i++)
10     {
11       float i=0;} /*début de vie du i flottant */
12       i=2*i;      /* le i flottant "cache" le i entier */
13     };            /*fin de vie du i flottant */
14   return (2*i);   /* i représente de nouveau le i entier */
15 }
```

FIG. 2.4 – Fonction C avec conflit de variables

Dans la section suivante, nous allons nous attarder sur une des difficultés du langage C très connue : la notion de pointeur.

2.5.3 Pointeurs

Une adresse de variable en langage C est considérée comme une valeur. Cette valeur est constante, car en général une variable C ne se déplace pas en mémoire et son adresse ne peut être modifiée par l'utilisateur. Il existe en langage C un type de variables permettant de manipuler les adresses d'autres variables : il s'agit des pointeurs.

Un **pointeur** est une variable qui contient l'adresse d'un autre variable. On dit que le pointeur pointe sur la variable pointée i.e. sur la variable dont il manipule l'adresse. Ici, pointer signifie «faire référence à».

Remarque(s) 1
La valeur d'un pointeur peut changer : cela ne signifie pas que la variable pointée est déplacée en mémoire mais plutôt que le pointeur pointe sur une autre variable, variable telle que nous l'avons défini dans la section 2.2.

Afin de savoir ce qui est pointé par un pointeur, ceux-ci disposent d'un type particulier.

Les pointeurs se déclarent en donnant le type de l'objet qu'ils devront pointer, suivi de leur identificateur précédé d'une étoile.

Illustration 5
La déclaration `"int *pi ;"` *est la déclaration d'une variable pointeur* `pi` *ne pouvant référencer que des variables de type entier.*

Il est possible de faire un pointeur sur n'importe quel type (types de bases, types structurés, types pointeurs). Un pointeur sur une variable indique la possibilité d'accéder à l'adresse de cette variable. Un pointeur permet d'accéder à la variable référencée par le pointeur via la valeur de son adresse.

Ces deux opérations sont respectivement appelées l'**indirection** et le **déréférencement**. Il existe deux opérateurs permettant de récupérer l'adresse d'un objet et d'accéder à l'objet pointé. Ces opérateurs sont respectivement `&` et `*`.

Illustration 6
Si nous regardons le code suivant :

```
int x;
int *p;
p=&x;
*p=12;
```

On commence tout d'abord par déclarer une variable `x` et un pointeur d'entier `p`. L'instruction `"p=&x;"` consiste à faire pointer `p` sur `x` en lui affectant l'adresse de `x` désignée par la notation `&x`. La valeur de `x` est modifiée par l'instruction `"*p=12;"` qui consiste à affecter la valeur 12 à la variable pointée par `p` soit à la variable `x`.

Les pointeurs permettent de modifier la valeur de la variable pointée via un autre identifiant de variable ce qui nous amène à introduire une nouvelle notion celle de variables synonymes ou alias. Des **variables synonymes ou alias** sont des variables désignant une même adresse mémoire. Il s'agit donc de variables différentes pointant la valeur d'une même case mémoire.

Illustration 7
Si on reprend l'illustration 6 précédente, la variable `x` et la variable référencée par le pointeur `p` sont des alias.

Pour le moment, nous avons écarté les pointeurs de notre analyse.

2.6 Fonctions et procédures en langage C

Nous avons déjà utilisé le terme de fonction C précédement pour désigner une routine C. Un programme C est composé d'un ensemble de routines calculant une sous-partie de l'ensemble du problème traité par la programme. Il s'agit d'un abus de langage dans la mesure où le terme de fonction désigne un type particulier de routine. Les deux types de routines en langage C se distinguent de la façon suivante : une fonction est une routine retournant une valeur d'un type défini dans sa déclaration et une procédure est une routine ne renvoyant pas de valeur particulière à l'exécution.

2.6.1 Procédures

Une **procédure** est une routine dont le type déclaré est le type C «`void`». Aucune valeur n'est associée à l'exécution d'une procédure.
La déclaration d'une procédure correspond au schéma suivant :

```
void ma_fonction(...)
{
 /*corps de la procédure*/
}
```

Cela ne signifie cependant pas que les procédures ne possèdent pas de variables de sortie et ne modifient pas le flot de contrôle de la routine dans laquelle elles sont appelées. En effet, elles peuvent modifier la valeur de certaines variables modifiables (comme la valeur d'une variable globale) qui peuvent être considérées comme des variables de sortie de la procédure. Nous reparlerons de cela plus précisément dans la section 2.6.5.

2.6.2 Fonctions

Le second cas concerne une routine C dont le type est différent de `void` et dont la déclaration correspond au schéma suivant :

```
un_type ma_fonction (...)
/* un_type est soit un type de base soit un type complexe*/
{
 /*corps de la fonction*/
 return(exp);/*exp est la valeur retournée du type un_type*/
}
```

Ce type de routine retourne une valeur, celle correspondant à l'évaluation de l'expression `exp` contenue dans l'instruction "`return(exp);`" et cette valeur doit être de même type que celui déclaré pour la fonction à savoir pour l'exemple `un_type`. Cette valeur est donc la valeur associée à l'exécution de la fonction.

Une **fonction** est donc une routine d'un type `un_type` autre que «`void`» retournant une valeur de type déclaré `un_type` à la fin de son exécution.

Remarque(s) 2
Pour préciser que certaines routines du langage C peuvent retourner une valeur, nous avons distingué dans cette section les fonctions et procédures du langage C. Cependant, par la suite, nous utiliserons le terme fonction pour désigner de façon générale une routine que ce soit une procédure ou une fonction C sauf précision de notre part.

Dans les schémas précédents représentant la syntaxe des fonctions et procédures, nous avons intentionnellement éludé la question des arguments d'une fonction en utilisant la notation suivante par exemple : "`un_type ma fonction (...);`". Nous allons maintenant définir ces arguments de fonction aussi nommés les paramètres d'une fonction[3].

2.6.3 Paramètres effectifs et paramètres formels d'une fonction

Les paramètres formels d'une fonction sont déclarés en même temps que celle-ci sous forme d'une liste de déclaration de variables entre parenthèses et accolée au nom de la fonction.

Par exemple, si les variables `param1` et `param2` toutes deux de type C `int` sont les paramètres de la procédure `fonc` alors la déclaration de la procédure `fonc` est de la forme suivante :

```
void fonc(int param1, int param2)
{
 /*corps de la fonction*/
}
```

Expression d'appel

Tout d'abord, attardons nous sur une autre expression du langage C : l'**expression d'appel**, notion qui sera essentielle dans notre approche de test structurel en présence d'appels de fonctions.

Cette expression consiste à utiliser le calcul d'une autre fonction (la fonction appelée) dans le flot de contrôle d'une fonction appelante. Une fois la fonction appelée et l'instruction contenant

[3] Nous rappelons que, par la suite, le terme de fonction désignera de façon générique toute routine C.

cette expression d'appel évaluée, l'exécution de la fonction appelante est poursuivie avec le résultat de l'expression d'appel. L'expression d'appel s'écrit en utilisant le nom de la fonction utilisée suivi d'une liste de paramètres appelée la liste des paramètres effectifs d'appel.

Illustration 8
Reprenons la procédure `fonc` *dont nous venons de donner la syntaxe. Une expression d'appel associée de cette fonction est de la forme* `"fonc(exp1,exp2);"` *avec* `exp1` *la première expression effective d'appel de la fonction* `fonc` *et* `exp2` *la seconde, toutes deux typées par* `int`*.*

Les expressions d'appel font partie des instructions à effets de bord dans la mesure où elles peuvent modifier la valeur de certaines variables de la fonction appelante. Après prétraitement du code source d'une fonction, une instruction contiendra au plus une expression à effets de bord (ce qui induit l'introduction de nouvelles variables). Par voie de conséquence, toute instruction après prétraitement contiendra au plus une expression d'appel et aucune autre expression à effets de bords.

Nous venons d'illustrer le cas d'un appel de procédure (pour laquelle aucune valeur n'est associée à l'exécution). Dans le cas d'un appel de fonction, l'expression d'appel ne se contente pas d'utiliser le calcul de la fonction appelée, elle affecte généralement le retour de la fonction ou une expression sur le retour de la fonction à une variable de la fonction appelante de même type déclaré pour la fonction appelée. Notons qu'il est possible que le retour de la fonction soit non utilisé.

Illustration 9
Prenons cette fois une fonction `fonc` *de type* `int` *dont la déclaration est de la forme :*

```
int fonc(int param1, int param2)
{
  /*corps de la fonction*/
  return(exp);
}
```

alors une expression d'appel de cette fonction correspond à une instruction de la forme `"var = fonc(exp1,exp2);"`*.*

Notons que dans notre cadre, une expression d'appel ne peut être contenue, après prétraitement, dans une instruction de la forme `"var = fonc(exp1,exp2)+1;"` *avec* `exp1` *la première expression effective d'appel,* `exp2` *la seconde dans la mesure où le prétraitement va isoler dans une instruction l'expression d'appel et dans une autre instruction l'incrémentation de la variable de retour de la fonction de la façon suivante :*

```
tmp=fonc(exp1,exp2);
var=tmp+1;
```

La variable `var` *est une variable de type* `int` *de la fonction appelante prenant la valeur de l'expression* `exp` *retournée par* `fonc` *incrémentée de 1.*

La sémantique de l'expression d'appel de la précédente illustration consiste tout d'abord à évaluer la fonction `fonc` avec pour valeurs en entrée les valeurs des expressions effectives d'appel puis d'affecter à la case mémoire identifiée par `var` la valeur de retour de la fonction `fonc`.

Paramètres formels versus paramètres effectifs

Pour illustrer plus en détail ces notions de paramètres formels et effectifs, nous allons développer l'exemple d'une fonction `f` appelant la fonction `g`. Les figures 2.5 et 2.6 donnent respectivement le code source des fonctions `f` et `g`. Notons que ces codes sources ne calculent rien de particulier et servent simplement d'illustration.

```
1    int   f(int x1,int   x2)
2    {
3      x1=  x2+x1;
4      if  (x1>10)
5         x1=(3*x1);
6      else
7         x2=x1;
8      x1=g(x1,x2);
9      return(x1);
10   }
```

FIG. 2.5 – Code source de la fonction f

```
1    int g(int z1,int z2)
2    {
3      z1= z2+z1;
4      z1= 2*z2;
5      z2= z1+z2;
6      return(z2);
7    }
```

FIG. 2.6 – Code source de la fonction g appelée dans la fonction f

Nous remarquons que la fonction `f` contient une instruction d'appel de la fonction `g` : `"x1=g(x1,x2);"` Cette instruction peut paraître ambiguë si on regarde les codes sources des deux fonctions. Il faut, en effet, bien distinguer les variables `z1` et `z2` dans la figure 2.6 qui désignent les paramètres formels de la fonction `g` et les variables `x1` et `x2` de l'expression d'appel de `f` dans la figure 2.5 qui sont les paramètres effectifs de `g` dont les valeurs effectives d'appel vont être affectées aux paramètres formels de `g` i.e. aux variables `z1` et `z2`.

Remarque(s) 3
Notons ici que les listes seront notées par des séquences entre [] dont les éléments sont séparés par des virgules.

Notation 1
La notation $|L|$ avec L une liste d'éléments représente la fonction de taille d'une liste, retournant le nombre d'éléments de la liste L.

Notation 2
Soit L une liste d'éléments et t le nombre de ses éléments tel que $t = |L|$, nous désignerons par $elem(L, i)$ la fonction retournant le i^{eme} élément de la liste L avec $0 \leq i \leq t$.

La liste ordonnée des **paramètres formels** d'une fonction C, notée $\mathcal{P}_{form}$, est la liste des arguments déclarés de la fonction dans l'implantation pour décrire le corps de la fonction.

Comme nous l'avons déjà suggéré précédemment, une expression d'appel n'est pas obligatoirement exécutée avec des variables comme paramètres effectifs mais peut être exécutée avec des expressions.

La liste ordonnée des **paramètres effectifs** d'une fonction, notée $\mathcal{P}_{eff}$, est la liste des expressions de variables utilisées dans une expression d'appel donnée (une ou plusieurs des expressions pouvant être réduites à une variable). Les valeurs des paramètres effectifs obtenues par évaluation sont affectées aux paramètres formels de la fonction appelée pour que celle-ci puisse effectuer son calcul.

Illustration 10
Si nous reprenons les codes sources des fonctions `f` *et* `g`*, nous avons donc :*
$\mathcal{P}_{eff} =$ [`x1,x2`]
$\mathcal{P}_{form} =$ [`z1,z2`]
ce qui correspond à l'appel suivant : `"g(x1,x2);"`

Pour une même fonction, les listes $\mathcal{P}_{eff}$ et $\mathcal{P}_{form}$ possèdent nécessairement le même nombre d'éléments donc vérifient

$$|\mathcal{P}_{eff}| = |\mathcal{P}_{form}|$$

et aussi que les éléments de même rang dans ces deux listes doivent être de même type.

Illustration 11
Reprenons la fonction `g` *précédente (cf. figure 2.6) appelée via l'instruction suivante* `"x1=g(x1+x2,x2-x1);"`*. La liste des paramètres formels ne change pas :* $\mathcal{P}_{form} =$ [`z1,z2`]*. En revanche, la liste des paramètres effectifs devient* $\mathcal{P}_{eff} =$ [`x1+x2, x2-x1`]*. De façon explicite, la valeur de l'expression* `"x1+x2"` *est affectée au paramètre formel* `z1` *et la valeur de l'expression* `"x2-x1"` *est affectée au paramètre* `z2` *et ce, avant le calcul de la fonction* `g`*.*

De l'affectation des paramètres formels d'une fonction

Il s'agit donc lors de l'appel de g de substituer aux variables contenues dans $\mathcal{P}_{form}$ les expressions de variables associées dans $\mathcal{P}_{eff}$ c'est-à-dire de procéder à l'affectation suivante :

$$elem(\mathcal{P}_{form}, i) = elem(\mathcal{P}_{eff}, i)$$

pour i allant de 1 à $|\mathcal{P}_{eff}|$ et ce, avant de lancer l'exécution de la fonction g.

Cela correspond donc simplement à une affectation séquentielle des paramètres formels de la fonction c'est-à-dire avant l'évaluation de la fonction appelée, n instructions d'affectations de la forme $elem(\mathcal{P}_{form}, i) = elem(\mathcal{P}_{eff}, i)$; sont évaluées de façon séquentielle avec $n = |\mathcal{P}_{eff}|$ ce qui correspond à la stratégie mise en place par le langage C.

Remarque(s) 4
Par construction $elem(\mathcal{P}_{form}, i)$ *est une variable tandis que* $elem(\mathcal{P}_{eff}, i)$ *est une expression.*

Dans la mesure où les paramètres formels sont juste déclarés et non définis, il ne peut exister de relation d'alias entre deux paramètres formels avant l'affectation des valeurs des paramètres effectifs. Il peut cependant exister entre deux expressions effectives d'appel une relation d'alias ainsi après l'affectation séquentielle des paramètres formels associés ceux-ci seront aussi liés par une relation d'aliasing et ce, quelque soit l'ordre des affectations des paramètres formels. Le prétraitement force l'ordre d'évaluation des paramètres effectifs d'appel d'une fonction.

Illustration 12
Prenons l'appel suivant `"f(i,i++)"` *pour la liste suivante des paramètres formels* $\mathcal{P}_{form} = [$`x,y`$]$. *Le prétraitement transforme cette instruction de la façon suivante :*

```
"tmp=i;
i=i+1;
f(tmp,tmp);"
```

En isolant les expressions à effets de bord, le prétraitement force l'évaluation des expressions effectives d'appel avant l'expression d'appel de la fonction. Ainsi, les paramètres formels `x` *et* `y` *prendront la même valeur lors de l'appel et la variable* `i` *est incrémentée de 1.*

La notion de pointeur ayant été définie dans la section 2.5.3, nous pouvons distinguer les notions de passage de paramètres par adresse (appelé aussi passage par référence) et de passage de paramètres par valeur.

2.6.4 Types de passage d'arguments

Il y a deux méthodes pour passer des variables en arguments dans une expression d'appel de fonction en langage C :
- le passage par valeur et
- le passage par adresse.

Lors d'un appel de fonction, chaque paramètre formel correspond à une variable locale de la fonction dont la durée de vie correspond au corps de la fonction appelée. Pour le **passage par valeur**, la valeur de l'expression passée en paramètre effectif est affectée à la variable locale correspondant au paramètre formel associé.

Ainsi, la valeur de l'expression passée par valeur est copiée dans une variable locale correspondant à la déclaration du paramètre formel associé. Aucune modification de la variable locale dans la fonction appelée ne modifie donc la variable passée en paramètre effectif.

Illustration 13
Le passage de paramètre par valeur correspond à l'exemple suivant :

```
void test(int j)    /* j stocke la copie de la valeur passée en
                       paramètre */
{
    j=3;            /* Modifie j, mais pas i. */
}

int main(void)
{
    int i=2;
    test(i);        /* Le contenu de i est copié dans j.
                       i n'est pas modifié. Il vaut toujours 2. */
    test(2);        /* La valeur 2 est copiée dans j. */
    return (i);     /* la valeur de i est retournée par la fonction */
}
```

Pour le **passage par adresse** appelé aussi **par référence**, l'adresse de la variable (ou de toute expression dont on peut calculer l'adresse) passée en paramètre effectif est affectée à la variable locale du paramètre formel associé, variable locale dont la déclaration stipule un type pointeur sur le type de l'expression effective d'appel associée.

Illustration 14
Le passage de paramètre par adresse correspond à l'exemple suivant :

```
void test(int *pj)  /* test attend l'adresse d'un entier... */
{
    *pj=2;          /* ... pour le modifier. */
    return 1;
}

int f()
{
    int i=3;
    test(&i);   /* On passe l'adresse de i en paramètre. */
    /* Ici, i vaut 2. */
    return 0;
}
```

Si on regarde le prototype de la fonction `test` alors on s'aperçoit que son paramètre formel attend l'adresse d'un entier.

Dans la fonction `f`, l'adresse de la variable `i` est passée par référence à la fonction `test` dans l'expression d'appel. Si on regarde maintenant le corps de la fonction `test`, son évaluation correspond à modifier la case mémoire dont l'adresse est contenue dans son paramètre formel en lui affectant la valeur 2.

Ce paramètre formel contient l'adresse `&i` ce qui signifie que la case mémoire identifiée par `i` est affectée par la valeur 2 lors de l'évaluation de la fonction `test`.

Un passage par adresse permet ainsi d'utiliser directement la valeur de l'expression effective associée pour faire les calculs (en déréférençant la variable locale du paramètre formel) dans la fonction appelée et aussi de conserver les éventuelles modifications de cette expression en manipulant directement la valeur stockée à l'adresse mémoire fournie. Les références et les pointeurs fonctionnent sur le même principe. En effet, si l'on utilise une référence pour manipuler un objet, cela revient exactement à manipuler un pointeur constant contenant l'adresse de l'objet manipulé. Les références permettent simplement d'obtenir le même résultat que les pointeurs avec une plus grande facilité d'écriture.

Ainsi, pour une expression d'appel avec passage par valeur de la forme `"fonc(&arg);"` et entraînant l'exécution d'une fonction de prototype `"void fonc(type_arg * arg_form)"` cela consiste, après le calcul de fonction appelée, à faire l'affectation suivante : `"arg=*arg_form;"`.

Les arguments passés par adresse sont affectés de la valeur déférencée des paramètres effectifs associés. Tout argument passé par référence dans une fonction et redéfini dans le corps de cette fonction est considéré comme une sortie.

Pour une expression d'appel avec plusieurs passages par adresse, il s'agit, en sortie de la fonction appelée, de procéder à une affectation multiple de ces arguments. L'ordre d'affectation n'est pas sans importance (contrairement à l'affectation multiple des arguments formels de la fonction appelée).

Illustration 15
Prenons l'implantation suivante :

```
void g(int *x1,int *x2, int x3)
{
  *x1=2+x3;  /*définition de la variable pointée par x1*/
  *x2=10;    /*définition de la variable pointée par x2*/
}
int f(int x)
{
  g(&x,&x,3); /*double passage par adresse de x et passage par valeur de 3 */
  if(x==10)   /*conditionnelle sur la valeur de x*/
    return(1);
  else
    return(0);
}
```

Les paramètres formels `x1, x2` de la fonction deviennent des alias de par l'expression d'appel qui leur affecte la même adresse de variable à savoir `"&x"`, ce qui signifie que les deux instructions de la fonction `g` modifient toutes deux la valeur de la variable `x` de la fonction `f`.

Le prétraitement force l'ordre d'évaluation des expressions de la fonction appelée à la sortie de l'appel de `g` dans `f` de la façon suivante : la variable `x` est affectée de la valeur 5 correspondant à l'évaluation de l'expression `2+x3` puis la variable `x` est affectée de la valeur 10. On s'aperçoit donc que la conditionnelle `if(x==10)` est toujours vérifiée et que donc la fonction `f` retourne toujours la valeur 1.

Notons que le prétraitement de la fonction sous test préserve la stratégie d'évaluation des expressions du langage C.

2.6.5 Variables d'entrée et variables de sortie d'une fonction C

Précisons maintenant ce que nous entendons par variables d'entrée et de sortie pour une fonction C.

Définition – 2.6.1

> *Les **variables d'entrée** d'une fonction `f` constituent l'ensemble des variables de la fonction utilisées avant d'être définies (au sens de l'utilisation et la définition d'une variable vue dans la section 2.3.1). Les variables d'entrée d'une fonction correspondent à un sous-ensemble des paramètres formels et/ou des variables référencées par ces paramètres formels et/ou des variables globales du programme.*

Nous représenterons l'ensemble de ces variables d'entrée comme les composants du vecteur d'entrée, X, de la fonction `fonc`.

Définition – 2.6.2

> *Les **variables de sortie** d'une fonction `fonc` constituent l'ensemble des variables définies par la fonction et dont la portée s'étend au delà de cette fonction ainsi que l'éventuelle variable de retour de celle-ci.*

Nous représenterons l'ensemble des variables de sortie comme les éléments du vecteur de sortie, Y, de la fonction `fonc`.

Si la fonction `f` est une fonction C et non une procédure (cf. section 2.6), le retour de la fonction noté $\mathcal{R}$(`f`) pour une fonction `f` est un singleton contenant la valeur de retour f_{return} de

la fonction `f`. Nous avons donc $\mathcal{R}(\texttt{f}) = \{\texttt{f\_return}\}$
Pour une procédure `f`, $\mathcal{R}(\texttt{f})$ sera donc vide.

Illustration 16
Soit l'expression d'appel `"x=g(x1,y+x);"`. *Nous avons donc :*
$\mathcal{P}_{eff} = [\texttt{x1, y+x}]$ *et* $\mathcal{R}(g) = \{\texttt{g\_return}\}$
avec `g_return` *contenant la valeur associée à cette exécution de* `g`. *Après exécution de cette fonction, la valeur de cette variable de retour* `g_return` *est affectée à la variable* `x` *de la fonction appelante.*

Notons que d'après la caractérisation faite des variables d'entrée et de sortie d'une fonction, il peut exister des variables de la fonction étant à la fois un élément de X et un élément de Y mais que leurs valeurs associées dans X et Y sont par conséquent différentes (puisqu'il y a eu redéfinition de ces variables). Ainsi, une variable peut être à la fois une variable d'entrée et de sortie de la fonction.

Notation 3
Si une variable `x` *est à la fois un élément de* X *et de* Y, *nous distinguerons sa valeur en entrée et en sortie de la fonction par la notation suivante :*
- `x` *représente sa valeur en sortie et*
- `x_in` *sa valeur en entrée de la fonction.*

L'exemple suivant, basé sur la figure 2.7, reprend une partie des notions de variables C que nous venons de caractériser.

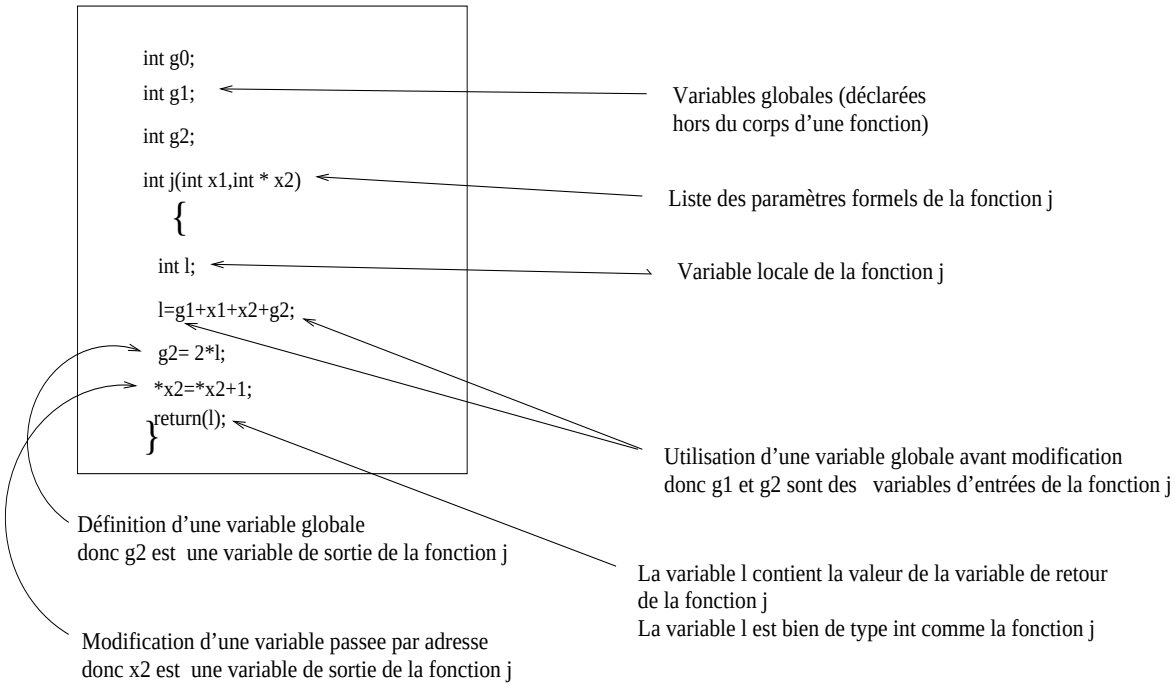

FIG. 2.7 – Illustration des différents types de variable C

Illustration 17
Dans la figure 2.7 se trouve une implantation d'une fonction `j` *en langage C avec l'identification des variables C présentes dans le corps de la fonction.*

Les variables `x1` *et* `*x2` *sont les paramètres formels de la fonction tels que* $\mathcal{P}_{form} = [$`x1,*x2`$]$ *et il s'agit ici d'un passage par valeur et d'un passage par adresse donc* `x2` *contient l'adresse d'une variable de la fonction appelante. Nous avons trois variables globales déclarées hors du corps de la fonction :* `g0,g1,g2`*. La variable* `l` *est déclarée dans le corps de la fonction j, il s'agit donc d'une variable locale à la fonction.*
Les variables globales `g1, g2` *étant utilisées avant toute définition dans la fonction* `j`*, il s'agit donc de variables d'entrée de la fonction* `j`*. Nous pouvons donc définir le vecteur des variables d'entrée comme :* $X =$ (`x1,*x2,g1,g2`)*. Nous observons aussi que la valeur de la variable globale* `g2` *est modifiée et donc qu'il s'agit aussi d'une sortie de la fonction ce qui impose de distinguer sa valeur initiale par l'introduction de la valeur* `g2_in`*. Nous remarquons aussi que la variable référencée par la variable* `x2` *est modifiée dans le corps de la fonction. Nous avons en fait* $X =$ (`x1,*x2,g1,g2`) *avec la variable* `g2` *valant* `g2_in` *en entrée de la fonction et la variable référencée par* `*x2` *vaut, de la même façon,* `*x2_in` *en entrée de la fonction.*
Nous construisons le vecteur de sortie de la fonction. La fonction `j` *est une fonction de type* `int` *donc* $\mathcal{R}($`j`$) =$ {`j_return`} *avec* `j_return` *de type* `int`*. Nous pouvons donc caractériser* Y *tel que :* $Y =$ (`g2,j_return,*x2`)*.*

Nous avons quatre variables d'entrée (X *possède 4 éléments) et trois variables de sortie (*Y *possède 3 éléments).*

Remarque(s) 5
L'ordre des variables dans les vecteurs X *et* Y *est arbitraire dans l'exemple.*

Pour introduire les chapitres suivants et en particulier le langage de spécification proposé des fonctions C, nous allons définir brièvement des définitions mathématiques associées aux fonctions C.

2.7 Introduction à notre modélisation mathématique des fonctions du langage C

Cette section va nous permettre d'introduire quelques unes des notations utilisées par la suite lors de la modélisation mathématique des fonctions du langage C.

Comme nous venons de le définir, X est le vecteur contenant les variables d'entrée d'une fonction et Y le vecteur contenant les variables de sortie d'une fonction. Ainsi pour une fonction d'implantation `f`, nous associons l'opération $f(X) = Y$.

Remarque(s) 6
Pour une fonction donnée :
- *f désignera la fonction décrite en langage naturel (par exemple la fonction addition),*
- `f` *désignera l'implantation de la fonction en langage C (par exemple* `int add(...)`
- *f la spécification de la fonction* ($f(x1, x2) = x1 + x2$).

Notation 4
Le symbole des fonctions partielles est $\rightarrow$.

Définition – 2.7.1

Une fonction f est un nom d'opération muni d'un profil

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

avec $s_{i[i=1..n]}$ et $s'_{j[j=1..m]} \in \mathcal{S}$ avec S l'ensemble des noms de types associés aux types C.

Notation 5

Pour la fonction f de profil :

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

on sera amené à utiliser les deux notations

$$f(x_1, \ldots, x_n) \rightarrow (y_1, \ldots, y_m)$$

$$f(X) \rightarrow Y$$

avec pour convention : x_i le i^{eme} élément de X une variable de sorte s_i et y_i le i^{eme} élément de Y une variable de sorte s'_i.

Cette notation sera particulièrement utile par la suite. Les noms des variables x_i d'entrée et y_i de sortie pourront également désigner les noms des variables d'affectation des entrées et des sorties dans l'implantation.

Les points les plus importants abordés dans ce chapitre concernent l'identification des variables d'entrée et de sortie d'une fonction C ainsi que la simplification du code source des fonctions effectuées lors de leur prétraitement.

L'utilité de ce prétraitement sera expliqué plus tard dans le chapitre 7.

La notion de variable d'entrée et de sortie d'une fonction est primordial pour l'utilisation de nos couples pre/post de notre langage de spécification. Le chapitre suivant présente ce langage de spécification dédié à la spécification des opérations des fonctions C.

Signalons que puisque le profil des opérations est de la forme

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

pour capturer le fait que les fonctions C peuvent avoir plusieurs variables de sortie, la modélisation mathématique des fonctions C ne peut être des fonctions simples avec un seul calcul (une seule sortie) pour résultat. Nous manipulerons donc des fonctions à codomaine multiple rendant comme résultat plusieurs valeurs.

Chapitre 3

Spécification des fonctions à base de formules pre/post

Dans ce chapitre, nous allons présenter la modélisation mathématique des fonctions du langage C mise en place pour notre langage de spécification.

Une **spécification d'une fonction** est une description du comportement de la fonction (détermination des paramètres et de leur type, de la sortie - ou des sorties de la fonction - , définition précise du calcul associé, ...). Notons qu'il y a de multiples notations plus ou moins formelles et complètes pour exprimer le comportement attendu d'une fonction comme la méthode B [Abr96], les spécifications algébriques [ST97], U.M.L [RJB04]....

Nous travaillons sur un langage de premier ordre [CL93] typé étendu car utilisant des codomaines multisortés dont la signature est $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P})$ avec $\mathcal{S}$ un ensemble de sortes, $\mathcal{V}$ un ensemble dénombrable de variables typées, $\mathcal{F}$ un ensemble d'opérateurs et $\mathcal{P}$ un ensemble de prédicats. Il s'agit plus précisément d'une logique du premier ordre.

3.1 Précisions sur la valeur indéfinie

Les fonctions partielles sont modélisées en étendant chaque sorte avec la valeur indéfinie $Undef$. La valeur $Undef$ est systématiquement propagée par application d'une opération : dès qu'un argument est évalué à $Undef$, le résultat est aussi $Undef$. Le prédicat $IsDef$ indique si un terme est défini ou non tel que :

$$IsDef(Undef) = false$$

$$IsDef(f(\ldots, Undef, \ldots)) = false$$

De façon générale, tous les prédicats seront définis ($true, false$). Un prédicat dont un argument est indéfini retournera $false$ pour éviter de rendre la satisfaction des formules plus difficile (cf. [ABK+02]).

Nous allons manipuler l'égalité dans notre langage de spécification, nous précisons qu'il s'agit d'une égalité dite forte : deux termes $t1$ et $t2$ sont égaux s'ils ont la même valeur et qu'ils sont tous deux définis tels que $IsDef(t1) = IsDef(t2) = true$.

Remarque(s) 7
Nous supposons que, les expressions manipulées sont, par construction, définies.

3.2 Sortes, opérateurs et formules de bases

Nous supposons disposer des sortes (types) de base coïncidant avec les types prédéfinis du langage C pour lesquels nous connaissons les domaines finis associés. En effet, dans la pratique,

pour chaque sorte de base, son domaine est fini et peut être associé à l'intervalle de valeurs représentables sur une machine du type C associé comme le montre le tableau 2.1 du chapitre 2, intervalles munis d'une relation d'ordre total.

Remarque(s) 8
Notons que nous traitons pour l'instant que les réels d'ECLiPSe [WNJ97]. La difficulté de traiter correctement cette représentation des réels est en grande partie liée aux propriétés mathématiques extrêmement pauvres des flottants : des propriétés comme l'associativité ou la distributivité ne sont pas vérifiées pour de nombreuses opérations et la structure des flottants n'est pas un corps. De plus le résultat d'un calcul dépend fortement des caractéristiques de l'unité de calcul. De nombreux travaux traitent de la question de la validation et de la vérification de programmes en présence de flottants [drV06]. Nous avons comme perspective à court terme d'intégrer les résultats des travaux de [BGM06] afin de pouvoir intégrer les flottants dans notre analyse.

Nous identifions un sous-langage $\mathcal{L}_0 = (\mathcal{S}_0, \mathcal{V}_0, \mathcal{F}_0, \mathcal{P}_0)$ de $\mathcal{L}$ regroupant les symboles (opérateurs ou prédicats) correspondant aux primitives du langage C à savoir les opérateurs $\mathcal{F}_0$, les prédicats $\mathcal{P}_0$ et les sortes primitives $\mathcal{S}_0$ extraits du langage C. Le sous-langage $\mathcal{L}_0$ contient, par exemple, les opérateurs spécifiant les symboles du langage C comme l'accès d'un champ d'une structure, l'accès à l'élément d'un tableau, ... Notons que, à des fins de spécification du langage C, nous utilisons aussi des opérateurs supplémentaires non présents en langage C mais indispensables ici comme l'opérateur de tableau retournant le nombre d'éléments de celui-ci.
Notons également que nous utilisons les quantificateurs (universel et existentiel) mais en limitant leur utilisation sur les seules sortes de $\mathcal{S}_0$ car il s'agit d'intervalles finis munis d'une relation d'ordre total.

Définition – 3.2.1

Nous disposons d'un ensemble de ***sortes primitives*** $\mathcal{S}_0$ *extraites du langage traité (le code C) tel que*

$$\mathcal{S}_0 = \{int, short, long, ushort, ulong, char, uchar\}$$

et pour tout s de $\mathcal{S}_0$, $Dom(s)$ est le domaine de s tel que

$$Dom(s) = [Min_s, Max_s]^1 \cup \{Undef_s\}$$

ce qui correspond à un ensemble fini de valeurs de type s. De plus on a :

$$\forall s_1 \in \mathcal{S}_0, \forall s_2 \in \mathcal{S}, (s_1 \neq s_2) \Rightarrow (Dom(s_1) \cap Dom(s_2)) = \emptyset$$

Le dernier point de la définition signifie que chaque valeur correspond à un unique type. Il est d'ailleurs parfois nécessaire d'expliciter, en pratique, le type associé. En effet, si on prend les sortes de base *int* et *long*, les supports associés sont $Dom(int) = [-2^{31} : int; 2^{31} - 1 : int] \cup \{Undef_{int}\}$ et $Dom(long) = [-2^{31} : long; 2^{31} - 1 : long] \cup \{Undef_{long}\}$.

Remarque(s) 9
Notons que comme les domaines sont finis, des dépassements de valeurs sont possibles. Travaillant dans le monde des domaines finis nous prenons en compte les calculs modulo la taille de l'intervalle comme pour les fonctions C.

[1] $[Min_s, Max_s]$ est l'intervalle de valeurs de type s avec Min_s (resp. Max_s), la plus petite (resp. grande) valeur représentable de l'intervalle.

Définition – 3.2.2

*Nous disposons d'un ensemble d'**opérateurs de base** typés, noté $\mathcal{F}_0$, portant sur l'ensemble $\mathcal{S}_0$. Chaque opérateur est muni d'un arité sur $\mathcal{S}_0^* \times \mathcal{S}_0^*$ et peut être éventuellement annoté pour éviter toute surcharge. L'ensemble $\mathcal{F}_0$ contient les opérateurs prédéfinis du langage C ainsi que des opérateurs supplémentaires nécessaires pour la spécification des fonctions C :*

$$\mathcal{F}_0 = \{+_{int\times int}; +_{int\times short}; \ldots; -_{int\times int}; \ldots *_{int\times int}; \ldots; \%_{int\times int}; ...\}$$

Remarque(s) 10

Le typage des fonctions C est de façon logique $w \rightarrow w'$ avec $w \in \mathcal{S}_0^*$ et $w' \in \mathcal{S}_0^*$. Nous pouvons donc avoir une fonction ne possédant pas de variable d'entrée à de multiples variables d'entrée et il en est de même pour les variables de sortie.

Comme nous l'avons vu auparavant, les fonctions contrairement aux procédures du langage C peuvent retourner une valeur, appelée valeur de retour de la fonction. De plus, dans la section 2.6.5, nous avons défini les variables de sortie d'une fonction C comme les variables dont les modifications sont conservées en sortie de la fonction y compris l'éventuelle variable de retour. Plusieurs sorties sont donc possibles pour une fonction en langage C.
Dans un tel cas de figure, habituellement pour une fonction C, la spécification associée peut-être décomposée en plusieurs parties, chaque partie correspondant à une sortie de la fonction comme dans l'exemple suivant.

Nous désirons n'avoir qu'une seule fonction de spécification pour chaque fonction C afin de rester le plus proche du langage testé. Notre langage de spécification contiendra donc des fonctions à codomaines multisortés. Deux choix s'offrent alors à nous :

- utiliser la projection pour isoler les différentes sorties de nos fonctions ou
- composer les multidomaines.

D'un point de vue technique, la première optique est la plus simple à mettre en place et donc celle choisie pour notre méthode.

Notation 6

Nous noterons notre opérateur de projection p_i^n représentant ici la i^{eme} projection d'un n-uplet. n étant implicite dans les cas évidents, nous surchargerons l'opérateur p en simplifiant la notation p_i^n par p_i.

Illustration 18

Ainsi, imaginons que nous avons la fonction f de profil :

$$f : int \rightarrow int \times int$$

et la fonction g de profil :

$$g : int \times int \rightarrow int$$

L'application de la fonction g aux sorties de la fonction f s'écrit donc :

$$g(p_1(f(x)), p_2(f(x)))$$

Introduisons de nouvelles notions comme l'occurrence de variable dans une formule logique ou la notion de variable libre. Mais avant tout, précisons les concepts de terme, formule logique atomique, formule logique.

Définition – 3.2.3

*Un **terme** est une suite de symboles construite à partir des règles suivantes :*

- *$true$ et $false$ sont des termes,*
- *si x est une variable de type s, alors x est un terme de type s,*
- *si c est une constante de type s, alors c est un terme de type s,*
- *si f est une fonction d'arité > 0 telle que $s_1 \times \cdots \times s_n \rightarrow s'_1 \times \cdots \times s'_m$ et que $t_1, \ldots, t_n$ sont des termes de type s_i, alors $f(t_1, \ldots, t_n)$ est un terme de type $s'_1 \times \cdots \times s'_m$.*
- *si t est un terme de sorte $s'_1 \times \cdots \times s'_m$ alors $p_k(t)$ est de sorte s'_k pour $k \leq m$.*

Définition – 3.2.4

*Une **formule logique atomique** est une suite de symboles construite à partir des règles suivantes :*

- *si $r \in \{<, \leq, \geq, >, =, \neq\}$ d'arité $n > 0$ avec $t_1, \ldots, t_n$ des termes avec t_i de type s_i alors $r(t_1, \ldots, t_n)$ est une formule logique atomique.*
- *si $pred \in \mathcal{P}$ est un prédicat d'arité $n > 0$ avec $t_1, \ldots, t_n$ des termes avec t_i de type s_i alors $pred(t_1, \ldots, t_n)$ est une formule logique atomique.*

Définition – 3.2.5

*Une **formule logique** est une suite de symboles construite à partir des règles suivantes :*

- *une formule atomique est une formule,*
- *si E est une formule, alors $\neg E$ est une formule,*
- *si $E1$ et $E2$ sont des formules, alors*
  - $E1 \wedge E2$
  - $E1 \vee E2$
  - $E1 \Rightarrow E2$
  - $E2 \Leftarrow E2$
  - $E1 \Longleftrightarrow E2$

  sont des formules,
- *si x est une variable et E est une formule, alors*
  - $\forall x E$
  - $\exists x E$ *sont des formules*
- *si x est une variable et $s \in \mathcal{S}$ le type associé alors*
  - $x \in s$

  est une formule

Quand une variable x apparaît dans une formule E, on dit qu'il y a une **occurrence** de x dans E et les différents endroits où apparaît la variable x dans la formule E sont nommés les différentes occurrences de x dans E.

Définition – 3.2.6

*Les occurrences d'une variable x dans une formule E sont dites des **occurrences libres** dans les cas suivants :*

- *E est atomique et contient la variable x,*
- *E est de la forme $\neg G$ avec G une formule et les occurrences de x sont libres dans G,*
- *E est de la forme $(G \alpha H)$ avec G et H des formules et α un connecteur binaire dans $\{\wedge, \vee, \Rightarrow, \Leftarrow, \Longleftrightarrow\}$ alors les occurrences libres de x dans E correspondent à celles de G ajoutées de celles de H et*
- *E est de la forme $(\exists y)G$ ou $(\forall y)G$ avec $x \neq y$ alors les occurrences libres de x dans G sont les occurrences libres de x dans E. Dans le cas où $x = y$ alors la variable x ne possède aucune occurrence libre mais uniquement des **occurrences liées**.*

Définition – 3.2.7

*Une variable x est dite **libre** dans une formule E si elle possède au moins une occurrence libre dans E.*

Notation 7

On notera l'ensemble des variables libres d'une formule E par $VarFree(E)$ et la notation $E(X)$ signifie que les variables libres de E sont incluses dans les composants de X tel que pour $E(X)$

$$VarFree(E) \subseteq X$$

3.2.1 Domaine et codomaine d'une fonction

Nous allons définir dans un premier temps la notion de domaine en entrée et en sortie d'une fonction f notés respectivement $Dom(f)$ et $CoDom(f)$.

Le domaine et le codomaine d'une fonction sont respectivement le produit cartésien des domaines associés aux données en entrée et le produit cartésien des domaines associés aux données en sortie de la fonction.

Définition – 3.2.8

*Le **domaine** de la fonction f de profil :*

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

*se définit comme : $Dom(f) = Dom(s_1) \times \ldots \times Dom(s_n)$ et son **codomaine** se définit comme : $CoDom(f) = Dom(s'_1) \times \ldots \times Dom(s'_m)$*

Illustration 19

Ainsi, le domaine de la fonction valAbs (fonction de calcul de la valeur absolue dont l'implantation `valAbs` *n'est pas précisée ici) de profil :*

$$valAbs : long \rightarrow ulong$$

est $Dom(valAbs) = Dom(long)$ et son codomaine est $CoDom(valAbs) = Dom(ulong)$.

3.2.2 Domaine de définition d'une fonction

Le domaine de définition d'une fonction noté $Def(f)$ correspond à l'ensemble des valeurs de $Dom(f)$ sur lequel la fonction f est définie c'est-à-dire possède une image spécifiée dans $CoDom(f)$:

$$Def(f) \subseteq Dom(f)$$

Définition – 3.2.9

***Le domaine de définition** $Def(f)$ est le sous-ensemble maximal de $Dom(f)$ pour lequel $f_{|Def(f)}$ est une fonction totale.*

Notation 8

On notera $f_{|X}$ la restriction de la fonction f au sous-ensemble X.

Notation 9

Le symbole des fonctions totales est $\mapsto$

Illustration 20
Pour la fonction valAbs, toutes les valeurs de $Dom(valAbs)$ ont une image dans $CoDom(valAbs)$ donc $Def(valAbs) = Dom(valAbs)$. Ainsi, la fonction valAbs est une fonction totale de $Dom(valAbs)$ sur $CoDom(valAbs)$ telle que :

$$valAbs : Dom(valAbs) \mapsto CoDom(valAbs)$$

Illustration 21
Prenons maintenant la fonction partielle div (fonction de calcul de la division entière d'un entier par un autre entier) de profil :

$$div : int \times int \rightarrow int$$

Toutes les valeurs de $Dom(div)$ n'ont pas une image dans $CoDom(div)$ (cas de la division par zéro) soit $Def(div) \subset Dom(div)$ et

$$Def(div) = (Dom(int) \setminus \{0\}) \times Dom(int)$$

3.3 Modèle associé

Nous considérons un seul modèle $\mathcal{M}_0$ fixé pour le langage $\mathcal{L}_0$ intuitivement celui qui interprète les symboles de $\mathcal{L}_0$ fidèlement par rapport au modèle opérationnel du langage C qui exécute les symboles de $\mathcal{L}_0$. Ce modèle $\mathcal{M}_0$ pourra être enrichi par l'utilisateur en définissant :

- $\mathcal{S}$ un ensemble de nouvelles sortes correspondant à un enrichissement de $\mathcal{S}_0$ par l'utilisateur,
- de nouveaux opérateurs (notons que l'utilisateur se devra dans ce cas de définir correctement cet opérateur et son domaine de définition pour maintenir un modèle unique)
- et par conséquent de nouvelles formules.

Pour définir les sortes associées aux types C standards comme les unions, les structures ..., nous introduisons les constructeurs de sortes suivants :

Définition – 3.3.1

Si s_1 et s_2 sont des sortes différentes de $\mathcal{S}$ et val_i,val_j des valeurs du même type $s \in \mathcal{S}$ avec $i, j \in I$ un ensemble d'entiers naturels,

- $\{val_i, ..., val_j\}$ *(énumérations)*
- $s1 \uplus$ [2] $s2$ *(unions)*
- $s \setminus \{val_i, ..., val_j\}$ *(avec val_i à val_j de type s) (sortes restreintes)*
- $s_1 \times s_2$ *(structures)*
- $s_1 \cap s_2$
- $s_1 \cup s_2$

sont des ***nouvelles sortes*** *de $\mathcal{S}$*

Nous noterons le modèle ainsi enrichi par l'utilisateur $\mathcal{M}$. Nous manipulerons par la suite $\mathcal{L}$ en partant du principe que la spécification est bien formée. Si ce n'est pas le cas alors pour tout ce qui aura été mal ou pas spécifié et mal ou pas défini, nous nous contenterons de lever une exception (message d'erreur). En effet, nous demandons à l'utilisateur de fournir une spécification détaillée, précise et correcte des opérateurs ou fonctions supplémentaires fournies.

$\mathcal{F}$ et $\mathcal{P}$ ont une unique interprétation sur $\mathcal{M}$ ainsi pour $\mathcal{M}_s = Dom(s)$

$$\mathcal{M} = ((\mathcal{M}_s)_{s \in \mathcal{S}}, (f_{\mathcal{M}})_{f \in \mathcal{F}}, (pred_{\mathcal{M}})_{pred \in \mathcal{P}})$$

avec $f_{\mathcal{M}}$ l'interprétation unique de la fonction f dans notre modèle telle que :

$$f_{\mathcal{M}} : \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n} \rightarrow \mathcal{M}_{s'_1} \times \ldots \times \mathcal{M}_{s'_m}$$

[2] $\uplus$ représente l'union exclusive : si la variable $x \in s1 \uplus s2$ alors x prend une valeur de $s1$ OU de $s2$: $s1 \uplus s2 = s1 \cup s2 \setminus s1 \cap s2$

pour toute fonction f de profil :

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

et $pred_{\mathcal{M}}$ définit un sous-ensemble de $s'_1 \times \ldots \times s'_m$

Définition – 3.3.2

*L'**interprétation d'une fonction** f sera notée $f_{\mathcal{M}}$ telle que :*

$$f_{\mathcal{M}} : Dom(f) \rightarrow CoDom(f)$$

$$f_{\mathcal{M}} : Def(f) \mapsto CoDom(f)$$

avec f une fonction totale de $Def(f)$ sur $CoDom(f)$.

Illustration 22

Le domaine de la fonction partielle div de profil :

$$div : int \times int \rightarrow int$$

est

$$Dom(div) = Dom(int) \times Dom(int)$$

et son codomaine est

$$CoDom(div) = Dom(int)$$

L'interprétation de la fonction div est donc notée :

$$div_{\mathcal{M}} : Dom(int) \times Dom(int) \rightarrow Dom(int)$$

La fonction div est une fonction totale de $Def(div)$ sur $CoDom(div)$ telle que :

$$div_{\mathcal{M}} : Def(div) \mapsto CoDom(div)$$

Notation 10

Nous notons ν l'interprétation des variables et $\mathcal{M} \models_\nu \phi$ la vérification de ϕ dans le modèle $\mathcal{M}$ selon l'interprétation ν.

3.4 Spécification des fonctions C

Nous avons choisi d'exprimer les spécifications des fonctions sous forme de couples de pre/post [Hoa69].

L'expression des spécifications sous forme pre/post a été privilégiée pour différents points :

- le coté simple d'appréhension d'un tel format d'expression (dans le sens caractérisation d'un sous-domaine en entrée associé à un comportement précis),
- la simplicité d'expression (nous aurions pu opter par exemple pour les spécifications algébriques mais nous les avons jugé plus complexes pour l'utilisateur) et
- le point de vue outillage (le rôle de chaque partie de contraintes des pre/post est clairement défini : valeurs interdites, caractérisation du sous-domaine en entrée, caractérisation entrées/sorties).

Le format des spécifications choisi manipule des formules de notre langage $\mathcal{L}$.

Avant d'aller dans le détail, nous allons donner deux illustrations du type de spécification attendue.

Illustration 23
Par exemple, prenons la fonction $valAbs(X) = Y$ *avec* $p_1(X) = x$ *et* $p_1(Y) = y$ *calculant la valeur absolue d'un nombre de profil :*

$$valAbs(x) = (y)$$

la fonction valAbs est totale
et deux sous-domaines en entrée de la fonction peuvent être distingués :

$$(x < 0)$$

$$(x \geq 0)$$

et pour chaque sous-domaine correspond un comportement précis de la fonction (en respectant l'ordre précédent de sous-domaines en entrée :

$$(y = -x)$$
$$(y = x)$$

Illustration 24
Si nous prenons encore l'exemple de la fonction

$$div(x_1, x_2) = (y)$$

la valeur nulle pour x_2 *est interdite :*

$$(x_2 \neq 0)$$

La fonction ne possède pas de comportement particulier pour de sous-domaines en entrée et le comportement en sortie de la fonction vérifie :

$$(y = x_1/x_2)$$

avec $p_1(X) = x_1$, $p_2(X) = x_2$ *et* $p_1(Y) = y$

Nous avons donc besoin de caractériser les contraintes de bonne utilisation d'une fonction, de caractériser les sous-domaines en entrée associé à un comportement donné de la fonction et enfin de caractériser ces comportements.

Chaque précondition d'une fonction représente un ensemble de contraintes (pouvant être des conjonctions et/ou disjonctions de contraintes) à respecter par l'utilisateur pour un bon fonctionnement de la fonction caractérisant le domaine de définition de la fonction et les postconditions les contraintes que la fonction doit remplir après exécution. Ce format à l'avantage d'être rapidement compréhensible par l'utilisateur, de plus il s'agit d'un format couramment utilisé qui permet une identification précise des changements d'états.

Définition – 3.4.1

Les préconditions *d'une fonction f de profil* $f(X) = Y$ *sont un ensemble de propriétés portant sur les variables de X. Nous noterons la conjonction des préconditions d'une fonction* $Pre(f, X)$*. Le domaine caractérisé par les préconditions se définit comme*

$$Dom(Pre(f, X)) = \{(a_1 \dots a_n) | \mathcal{M} \models_\nu Pre(f, X)\}$$

avec ν *interprétation unique sur* $\mathcal{M}$ *telle que* $\forall i, 1 \leq i \leq n, \nu(x_i) = a_i$ *avec* $x_i = p_i(X)$ *pour X à n composantes.*

Le domaine des préconditions correspond au domaine de définition de la fonction $Dom(Pre(f,X)) = Def(f)$.

Le non respect de l'ensemble des contraintes des préconditions d'une fonction correspond à une valeur non définie $Undef$ et l'application d'une fonction sur une valeur non définie correspond à un résultat non défini.

Les postconditions traduisent le **comportement attendu de la fonction** et permettent d'exprimer les propriétés sur les sorties prévues.
Nous avons choisi comme format d'expression des postconditions le format suivant :

$$(1)Post(f,X,Y) : (D(f,X) \wedge Q(f,X,Y))$$

car il est facilement compréhensible par l'utilisateur : $D(f,X)$ détermine un sous-domaine en entrée de la fonction et $Q(f,X,Y)$ définit le comportement attendu pour ce sous-domaine par l'expression des propriétés à vérifier pour X et Y. Il est toutefois possible que la postcondition soit exprimée selon le format suivant plus général :

$$(2)Post(f,X,Y) : Q(f,X,Y)$$

dans ce cas, les contraintes exprimées par $Q(f,X,Y)$ s'appliquent sur la totalité du domaine de définition de la fonction.

Définition – 3.4.2

Les postconditions *d'une fonction f telle que $f(X) = Y$ sont un ensemble de propriétés $Post(f,X,Y)$ sur les entrées et les sorties de f dont le domaine se définit comme :*

$$Dom(Post(f,X,Y)) = \{(a_1 \dots a_n) | \exists b_1 \dots b_m \mathcal{M} \models_\nu Post(f,X,Y)\}$$

avec ν interprétation unique sur $\mathcal{M}$ telle que

$$\forall i, 1 \leq i \leq n, \nu(x_i) = a_i$$

Notons que nous nous limitons ici aux cas où $Q(f,X,Y)$ est une expression fonctionnelle de Y en fonction de X i.e. qui n'induit pas de contraintes supplémentaires sur X par rapport à $D(f,X)$ soit :

$$Dom(Post(f,X,Y)) = Dom(D(f,X))$$

La seconde partie d'une postcondition, $Q(f,X,Y)$, définit le comportement attendu de la fonction dans le codomaine de la postcondition :

$$CoDom(Post(f,X,Y)) = \{(b_1 \dots b_m) | \exists a_1 \dots a_n \in Dom(Post(f,X,Y)), \mathcal{M} \models_\nu Post(f,X,Y)\}$$

avec ν interprétation unique sur $\mathcal{M}$ telle que

$$\forall i, 1 \leq i \leq n, \nu(x_i) = a_i$$

$$\forall j, 1 \leq j \leq m, \nu(y_j) = b_i$$

toujours avec $x_i = p_i(X)$ pour X à n composantes et $y_j = p_j(X)$ pour Y à m composantes.

Pour une fonction f telle que $f(X) = Y$, les préconditions $Pre(f,X)$ et chaque postcondition $Post_i(f,X,Y)$ peuvent être représentées par analogie au triplet de la logique de Hoare sous la forme :

$$Pre(f,X) \wedge D_i(f,X)[f(X) = Y]Q_i(f,X,Y)$$

avec $Post_i(f, X, Y) = (D_i(f, X) \wedge Q_i(f, X, Y))$. Le sens intuitif du triplet de Hoare précédent est que si X vérifie $Pre(f, X) \wedge D_i(f, X)$ alors f se termine et à l'issue de l'exécution de f , X et Y vérifient $Q_i(f, X, Y)$.

$Pre(f, X)$ et $Post_i(f, X, Y)$ sont des formules universellement quantifiées dans $\mathcal{L}$ et respectivement sur X et $X \cup Y$.

Nous noterons un couple pre/post $PP_i(f, X, Y)$ d'une fonction f telle que $f(X) = Y$

$$PP_i(f, X, Y) : Pre(f, X) \wedge D_i(f, X)[f(X) = Y]Q_i(f, X, Y)$$

$$PP_i(f, X, Y) = Pre(f, X) \wedge D_i(f, X), Q_i(f, X, Y)$$

Convention 1
Pour alléger un peu les notations, $Dom(f)_{|Pre(f,X)\wedge D_i(f,X)}$ *sera noté* $Dom(Pre(f, X) \wedge D_i(f, X))$.

Le support d'un couple pre/post $PP_i(f, X, Y)$, $Dom(PP_i(f, X, Y))$ et son codomaine, $CoDom(PP_i(f, X, Y))$ sont définis comme :

$$Dom(PP_i(f, X, Y)) = Dom(Pre(f, X)) \cap Dom(Post(f, X, Y))$$

$$= Dom(Pre(f, X)) \cap Dom(D_i(f, X)$$

$$CoDom(PP_i(f, X, Y)) = CoDom\{(b_1 \dots b_m) | \exists (a_1 \dots a_n) \in Dom(PP_i(f, X, Y)), \mathcal{M} \models_\nu Q_i(f, X, Y)\}$$

L'interprétation d'un couple pre/post dans notre modèle vérifie :

$$PP_i^{\mathcal{M}}(f, X, Y) =$$

$$\{(a_1 \dots a_n, b_1 \dots b_m) |$$

$$(a_1 \dots a_n) \in Dom(PP_i(f, X, Y)), (b_1 \dots b_m) \in CoDom(PP_i(f, X, Y)),$$

$$\mathcal{M} \models_\nu Pre(f, X) \wedge D_i(f, X) \wedge Q_i(f, X, Y)\}$$

avec $\nu(x_i) = a_i$ et $\nu(y_i) = b_i$.

La spécification de la fonction f, notée $Spec(f, X, Y)$ correspond à un ensemble fini de couples pre/post pour f :

$$Spec(f, X, Y) : \{PP_i(f, X, Y)\}_{i \in I}$$

avec I un ensemble d'entiers naturels borné.

Remarque(s) 11
Les domaines des différents couples pre/post d'une fonction f *sont aussi appelés les différents domaines fonctionnels de la fonction* f*. Il s'agit de sous-domaines des valeurs des variables d'entrée entraînant un même comportement de la fonction.*

Définition – 3.4.3

*Un **domaine fonctionnel** DF_i correspond au support d'un couple pre/post de la spécification tel que :*

$$DF_i = Dom(PP_i(f, X, Y))$$
$$PP_i(f, X, Y) \in Spec(f, X, Y)$$

3.5 Quelques propriétés des spécifications

Le but de notre langage de spécification est de caractériser de façon précise le comportement des fonctions appelées pour permettre de modéliser (et simuler) leur comportement pour la mise en place d'une méthode de gestion originale des appels de fonctions (nous verrons cela plus en détails dans la partie III) .

Il faut donc caractériser de façon très précise notre langage de spécification en lui fixant quelques restrictions, restrictions dont nous montrerons la nécessité lors de l'explication de notre stratégie.

Une des premières restrictions logiques est que la totalité du domaine de définition de la fonction sous test soit caractérisée dans notre spécification : il s'agit de la notion de complètude.

Définition – 3.5.1

*Soit $f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$ une fonction C, soit $Spec(f, X, Y)$ une spécification pour f. $Spec(f, X, Y)$ est une **spécification complète** sur son domaine de définition, $Def(f)$ si l'union de ses sous-domaines $D_i(f, X)$ recouvre le domaine des préconditions $Dom(Pre(f, X))$:*

$$Dom(Pre(f, X)) \subset \bigcup_{i=1}^{n} Dom(D_i(f, X))$$

pour $Spec(f, X, Y) : \{PP_i(f, X, Y)\}_{i \in I}$ et $D_i(f, X) \in PP_i(f, X, Y)$

Illustration 25

Soit la fonction valAbs retournant la valeur absolue d'un entier et de profil $valAbs : long \rightarrow ulong$, son domaine de définition

$$Def(valAbs) = Dom(long)$$

et sa spécification

$$Spec(valAbs, X, Y) = \{PP_1(valAbs, X, Y), PP_2(valAbs, X, Y)\}$$

avec $p_1(X) = x$ et $p_1(Y) = y$ et avec :

$$PP_1(valAbs, X, Y) : (true) \wedge (x < 0)[valAbs(x) = y](y = -x)$$

$$PP_2(valAbs, X, Y) : (true) \wedge (x \geq 0)[valAbs(x) = y](y = x)$$

La spécification est ici complète car elle vérifie la relation :

$$Def(valAbs) = \bigcup_{i=1}^{2} Dom(PP_i(valAbs, X, Y)) = Dom(long)$$

avec

$$Dom(PP_1(valAbs, X, Y))) = \{x | x \in Dom(long), x < 0\}$$
$$Dom(PP_2(valAbs, X, Y))) = \{x | x \in Dom(long), x \geq 0\}$$

Nous imposons l'absence de couples pre/post contradictoires qui introduirait une spécification incohérente. Nous désignons par couples pre/post contradictoires des couples pre/post dont les domaines ne sont pas disjoints mais dont les propriétés sur les variables de sorties sont contradictoires i.e. ne produisent pas les mêmes sorties. La présence de couples pre/post contradictoires dans une spécification entraînerait des comportements différents pour un même sous-domaine d'entrée. Ce qui ne nous permettrait pas de définir exactement le comportement réel de la fonction mais un ensemble de comportements possibles.

Définition – 3.5.2

Soit la fonction de profil :

$$f : s_1 \times \ldots \times s_n \rightarrow s'_1 \times \ldots \times s'_m$$

Deux couples pre/post $PP_1(f,X,Y), PP_2(f,X,Y)$ de sa spécification $Spec(f,X,Y)$ sont dits ***couples pre/post contradictoires*** *si la relation suivante est vérifiée :*

$$\exists((a_1,..,a_n),(b_1,..,b_m)) \in PP_1^{\mathcal{M}}(f,X,Y), \exists((a_1,..,a_n),(b'_1,..,b'_m)) \in PP_2^{\mathcal{M}}(f,X,Y),$$

$$(b'_1,..,b'_m) \neq (b_1,..,b_m)$$

La spécification $Spec(f,X,Y)$ est alors dite ***incohérente****.*

Par soucis de simplicité, nous imposons donc que les domaines des couples pre/post soient disjoints.

Cette restriction faite sur les spécifications d'une fonction concerne au moins 2 couples pre/post, une autre propriété imposée aux spécifications d'une fonction possède le même rôle à la différence près qu'elle ne s'applique qu'à un seul couple pre/post (toujours pour des domaines de pre/post disjoints) : il s'agit de la propriété de déterminisme.

Définition – 3.5.3

Une spécification $Spec(f,X,Y)$ est dite ***déterministe*** *si chaque couple pre/post $PP_i(f,X,Y) \in Spec(f,X,Y)$ vérifie la relation suivante :*

$$\forall(a_1,..,a_n) \in Dom(PP_i(f,X,Y)), \exists!(b_1,..,b_m), ((a_1,..,a_n),(b_1,..,b_m)) \in PP_i^{\mathcal{M}}(f,X,Y))$$

Illustration 26

Soit une fonction de profil $f : s_1 \times s_2 \rightarrow s'_1$ et sa spécification $Spec(f,X,Y) = \{PP(f,X,Y)\}$ telle que :

$$PP(f,X,Y) : ((x1 > 0) \wedge (x2 > 0)), (y > (x_1 + x_2))$$

avec $p_1(X) = x1$, $p_2(X) = x2$ et $p_1(Y) = y$. Le retour de la fonction correspond ici à un ensemble de valeurs possibles (un intervalle construit sur la valeur des variables de X) et non à une valeur unique en sortie. La spécification est donc non déterministe.

Illustration 27

La fonction `pgcd(x1,x2)` *de la figure 3.1 retourne le plus grand diviseur commun des variables* `x1` *et* `x2`*. Il s'agit d'une fonction partielle sur son domaine $Dom(pgcd)$ car elle n'est définie que pour des valeurs strictement positives de* `x1` *et* `x2`*.*

Si nous considérons la spécification suivante (qui est incomplète) :

$$Spec(pgcd,X,Y) = \{PP(pgcd,X,Y)\}$$

$$PP(pgcd,X,Y) = (Pre(pgcd,X) \wedge D(pgcd,X), Q(pgcd,X,Y))$$

```
1   int pgcd (int x1, int x2)
2  {
3    int q,r;
4    do
5     {
6       q = x1 / x2;
7       r = x1 - q*x2;
8       x1 = x2;
9       x2 = r;
10    }
11   while (x2 > 0);
12
13   return x1;
14 }
```

FIG. 3.1 – Code source de la fonction `pgcd`

$$Pre(pgcd, X) = (x1 > 0) \wedge (x2 > 0)$$

$$D(pgcd, X) = true$$

$$Q(pgcd, X, Y) = (min(x1, x2) \geq y \geq 1 \wedge (x1\%y = 0) \wedge (x2\%y = 0))$$

$p_1(X) = x1$, $p_2(X) = x2$ *et* $p_1(Y) = y$.

L'opérateur $min()$ *retourne la valeur minimale de ses arguments. L'opérateur* $\%$ *appelé modulo en langage C retourne le reste de la division de son premier argument par le second.*

La spécification nous donne ici les caractéristiques de la valeur de retour de la fonction mais ne permet pas de l'identifier exactement. En effet, si nous appliquons la spécification pour calculer le pgcd avec $x1 = 8$ *et* $x2 = 24$ *alors* y *peut prendre sa valeur dans l'ensemble* $\{1; 2; 4; 8\}$ *alors que le résultat exact est* $y = 8$ *ici. Pour que cette spécification soit déterministe, il aurait fallu ajouter que la solution exacte est la plus grande valeur de l'ensemble ce qui correspond à modifier la seconde partie de la postcondition de la façon suivante :*

$$Q(pgcd, X, Y) = (min(x1, x2) \geq y \geq 1 \wedge (x1\%y = 0) \wedge (x2\%y = 0)) \wedge$$

$$\neg(\exists y2((min(x1, x2) \geq y2 \geq 1 \wedge (x1\%y2 = 0) \wedge (x2\%y2 = 0)))) \wedge (y2 > y)$$

ce qui correspond à préciser que la sortie est la valeur la plus grande des valeurs possibles de l'ensemble précédent.

Remarque(s) 12
Nous excluons également de notre analyse les fonctions dont les spécifications sont récursives. Nous ne nous sommes pas penchés sur ce point pour l'instant mais cela fait partie des points envisagés comme une extension de la stratégie de gestion des appels de fonctions présentée dans ce manuscrit.

```
1  int tritype(int i, int j, int k) {
2    int trityp ;
3    if ((i == 0)||(j == 0)||(k == 0))
4       trityp = 4 ;
5    else {
6       trityp = 0 ;
7       if (i==j) trityp = trityp + 1 ;
8       if (i==k) trityp = trityp + 2 ;
9       if (j==k) trityp = trityp + 3 ;
10      if (trityp==0) {
11         if ((i+j <= k)||(j+k<=i)||(i+k<=j))
12           trityp = 4 ;
13         else trityp = 1 ;
14      }
15      else {
16         if (trityp>3) trityp = 3 ;
17         else if ((trityp==1)&&(i+j>k))
18                trityp = 2 ;
19              else if ((trityp==2)&&(i+k>j))
20                     trityp = 2 ;
21                   else if ((trityp==3)&&(j+k>i))
22                          trityp = 2 ;
23                        else trityp = 4 ;
24      }
25    }
26    return trityp;
27  }
```

FIG. 3.2 – Code source de la fonction `tritype`

Illustration 28
La fonction C `tritype` *prend en entrée 3 variables de type entiers de valeurs positives ou nulles représentant les longueurs des côtés d'un triangle et retourne :*

- *1 si le triangle en question est scalène,*
- *2 s'il est isocèle,*
- *3 s'il est équilatéral et*
- *4 s'il ne s'agit pas d'un triangle.*

La fonction $tritype(X) = Y$ *est de profil :*

$$tritype : Dom(int) \times Dom(int) \times Dom(int) \rightarrow Dom(int)$$

La spécification associée est la suivante :
$Spec(tritype, X, Y) = \{PP_1(tritype, X, Y), PP_2(tritype, X, Y), PP_3(tritype, X, Y), PP_4(tritype, X, Y)\}$
avec $p_1(X) = x_1$, $p_2(X) = x_2$, $p_3(X) = x_3$ *et* $p_1(Y) = y_1$

$Pre(tritype, X) = (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_3 \geq 0)$

$PP_1(tritype, X, Y) =$
$((Pre(tritype, X)) \wedge ((x_1 + x_2 \leq x_3) \vee (x_2 + x_3 \leq x_1) \vee (x_1 + x_3 \leq x_2)), y_1 = 4)$

$PP_2(tritype, X, Y) =$
$(\neg(D1(tritype, X)) \wedge Pre(tritype, X) \wedge ((x_1 = x_2) \wedge (x_2 = x_3)), y_1 = 3)$
avec
$D1(tritype, X) = (x_1 + x_2 \leq x_3) \vee (x_2 + x_3 \leq x_1) \vee (x_1 + x_3 \leq x_2)$

$PP_3(tritype, X, Y) =$
$(\neg(D1(tritype, X)) \wedge \neg(D2(tritype, X)) \wedge Pre(tritype, X) \wedge ((x_1 = x_2) \vee (x_2 = x_3)$
$\vee(x_1 = x_3)), y_1 = 2)$
avec
$D2(tritype, X) = (x_1 = x_2) \wedge (x_2 = x_3)$

$PP_4(tritype, X, Y) =$
$(\neg(D1(tritype, X)) \wedge \neg(D2(tritype, X)) \wedge \neg(D3(tritype, X)) \wedge Pre(tritype, X), y_1 = 1)$
avec
$D3(tritype, X) = (x_1 = x_2) \vee (x_2 = x_3)$

Avec $Pre(tritype, X)$*, nous pouvons déterminer le domaine de définition de la fonction comme*

$$Def(tritype) = [0, MaxInt] \times [0, MaxInt] \times [0, MaxInt]$$

Cette spécification répond à nos différentes restrictions :

- *la sortie attendue est ici caractérisée de façon unique (les valeurs sont données) donc il s'agit bien d'une spécification déterministe,*
- *chaque couple pre/post nie les conditions des autres couples pre/post garantissant l'absence de couples pre/post contradictoires,*
- *l'union des domaines des 4 couples pre/post couvre exactement le domaine de définition de la fonction ce qui signifie que la spécification est complète sur son domaine de définition.*

3.6 Spécification d'une fonction appelée

Tout ce que nous avons dit précédemment sur la spécification d'une fonction reste vrai pour des fonctions appelées ou des fonctions appelantes au sens du langage C.

Remarque(s) 13
Nous allons cependant préciser un point. La formulation des codes et de la spécification d'une fonction peuvent ne rien avoir en commun. En particulier, le code de la fonction `f` *peut s'exprimer (utiliser) le code de la fonction* `g` *sans pour autant que la spécification de la fonction* `f` *s'exprime en fonction de la spécification de la fonction* `g` *et vice-versa.*

Illustration 29
La figure 3.3 contient les codes sources des fonctions `doubleSomme` *et* `somme` *de profils* $doubleSomme : int \times int \rightarrow int$ *et* $somme : int \times int \rightarrow int$ *avec la fonction* `somme` *appelée dans la fonction* `doubleSomme`*. Les spécifications associées aux fonctions sont définies de la façon suivante :*

$$Spec(somme, X, Y) = \{PP(somme, X, Y)\} = \{(Pre(somme, X) \wedge D(somme, X), Q(somme, X, Y))\}$$

$$D(somme, X) = Pre(somme, X) = true$$

$$Q(somme, X, Y) = (p_1(Y) = p_1(X) + p_2(X))$$

$$Spec(doubleSomme, X, Y) = \{PP(doubleSomme, X, Y)\} =$$

$$\{(Pre(doubleSomme, X) \wedge D(doubleSomme, X), Q(doubleSomme, X, Y))\}$$

$$D(doubleSomme, X) = Pre(doubleSomme, X) = true$$

$$Q(doubleSomme, X, Y) = (p_1(Y) = 2 * (p_1(X) + p_2(X)))$$

Nous voyons ainsi que l'imbrication des codes sources ne se vérifie pas obligatoirement dans les spécifications.

```
1  int somme(int x1, int x2)
2  {
3    int r;
4    r = x1 + x2;
5    return r;
6  }
7
8  int doubleSomme(int x1, int x2)
9  {
10   int r;
11   r = 2* somme (x1, x2);/*instruction d'appel */
12   return r;
13 }
```

FIG. 3.3 – Exemple simple d'appel de fonction

De la même façon, certaines fonctions récursives (c'est-à-dire appelées par elles-mêmes) peuvent être spécifiées sans imbrication.

Illustration 30
Prenons la fonction `n_somme` *qui calcule la somme des n premiers nombres entiers de profil :*

$$n\_somme : int \mapsto int$$

une spécification possible est :

$$Spec(n\_somme, X, Y) = \{PP(n\_somme, X, Y)\} =$$

$$\{(Pre(n\_somme, X), D(n\_somme, X), Q(n\_somme, X, Y))\}$$

$$Pre(n\_somme, X) = (p_1^1(X) >= 0)$$

$$D(n\_somme, X) = true$$

$$Q(n\_somme, X, Y) = (p_1(Y) = (p_1(X) * (1 + p_1(X)))/2)$$

Sa spécification est non récursive mais son implantation peut l'être comme le montre la figure 3.4.

```
1  int n_somme(int x)
2  {
3    int r;
4    if (x==0)
5       r=0;
6    else
7       r=x+n_somme(x-1);/*instruction d'appel */
8    return r;
9  }
```

FIG. 3.4 – Implantation récursive de la fonction `n_somme`

Nous venons de voir ici l'utilisation des variables d'entrée et de sortie d'une fonction du côté spécification, ce qui justifie la distinction faite sur ces variables dans le chapitre 2. Notons que du côté des spécifications les variables d'entrée et de sortie sont nécessairement deux ensembles bien distincts.

Les spécifications de fonction sont assez simples mais donneront lieu à de nombreuses manipulations en terme de concrétisation dans la stratégie de gestion des appels de fonction expliquée au chapitre 11.

Chapitre 4

Graphe de flot de contrôle d'une fonction C

Pour représenter le flot de contrôle d'une fonction, nous utilisons une représentation par un graphe connexe orienté $G =< N, E, e, s, \delta >$ avec N l'ensemble des nœuds, E l'ensemble des arcs, un unique nœud d'entrée e et un unique nœud de sortie s, δ est la fonction d'étiquetage du graphe tel que les nœuds sont associés à des instructions de la fonction et les arcs à des conditions modifiant le flot de contrôle entre ses instructions.

L'annexe B contient toutes les notions et définitions de la théorie des graphes [Ber58] utilisées dans ce manuscrit.

4.1 Représentation des structures du langage C

Reprenons les différentes structures du langage C présentées dans le chapitre 2 et un graphe étiqueté $G =< N, E, e, s, \delta >$.

Nous allons ainsi définir la représentation des différentes structures du langage.

Commençons tout d'abord par le cas d'un ensemble d'instructions séquentielles simples (i.e. un ensemble d'instructions séquentielles ne modifiant pas le flot de contrôle) de la forme :

```
{
inst1;
inst2;
...
instN;
}
```

Une première propriété facilement observable concerne les ensembles d'instructions séquentielles simples que nous désignerons comme blocs de base du langage C.

Définition – 4.1.1

*Un **bloc de base** b est une séquence d'instructions consécutives contenant au plus une jonction au début (i.e. une arrivée de flot d'exécution) et un embranchement à la fin (un départ du flot d'exécution) et vérifiant :*

$$b = (n_1, (n_1, n_2), n_2, \ldots, (n_{k-1}, n_k), n_k)$$

$$\forall i, 1 \leq i \leq k, n_i \in N$$

$$\forall i, 1 \leq i \leq k-1, (n_i, n_{i+1}) \in E$$

$$\forall j, 2 \leq j \leq (k-1), |pred(n_j)| = |succ(n_j)| = 1$$

$$|succ(n_1)| = |pred(n_k)| = 1$$

avec N l'ensemble des instructions, E l'ensemble des flots d'exécution de N vers N et $pred(n_i)$ (resp. $succ(n_i)$) l'ensemble des arcs entrants (resp. sortants) du nœud n_i.

Convention 2

Pour alléger la représentation graphique, un bloc de base correspond à un unique nœud dans le graphe (cf. figure 4.1).

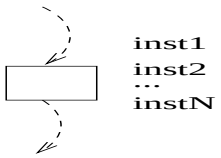

Fig. 4.1 – Représentation d'un bloc de base

Les figures 4.2, 4.3, 4.4 contiennent respectivement la représentation graphique des structures IfThenElse, IfThen, Switch, SwitchBreak et des structures conditionnelles dont les syntaxes ont été présentées dans les figures 2.1, 2.2, 2.3 du chapitre 2 et les figures A.1, A.2, A.3 et A.4 de l'annexe A où b est le bloc suivant la construction.

Remarque(s) 14

Un nœud fictif sans label est ajouté dans les figures 4.2, 4.3, 4.4 pour représenter la sortie des structures. Notons que ce nœud fictif n'est ajouté que pour la représentation de ces figures afin d'en faciliter la lecture mais qu'il n'existe pas en réalité selon la définition 4.1.1 des blocs de base d'une fonction.

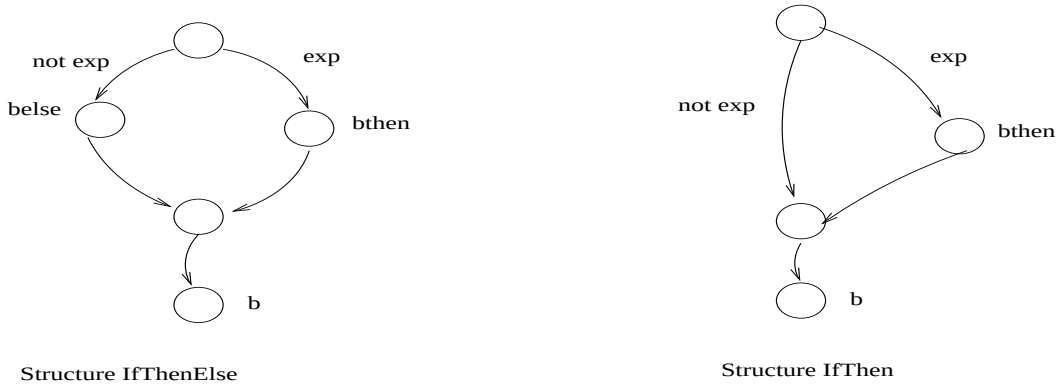

FIG. 4.2 – Représentation des structures IfThen et IfThenElse

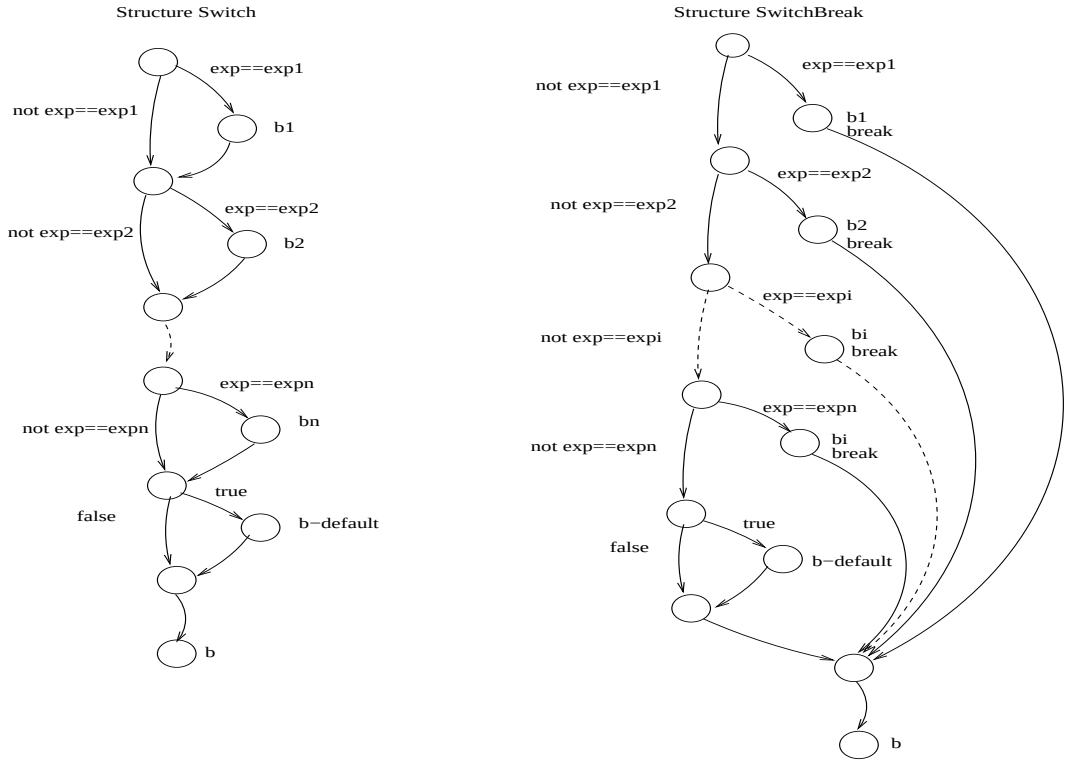

FIG. 4.3 – Représentation des structures Switch et SwitchBreak

4.2 Construction d'un CFG

Ainsi, le CFG construit à partir du code source d'une fonction est un graphe dont les arcs représentent l'ensemble des branchements introduisant un nouveau flot de contrôle dans la fonction et les nœuds représentent les blocs de base de la fonction associée.

Par branchement du graphe, on entend ici soit une structure conditionnelle soit une structure répétitive (voir les représentations graphiques de ces instructions dans les figures 4.4, 4.3, 4.2).

Pour construire le CFG d'une fonction, il s'agit donc d'identifier dans l'implantation associée tous les blocs de base.

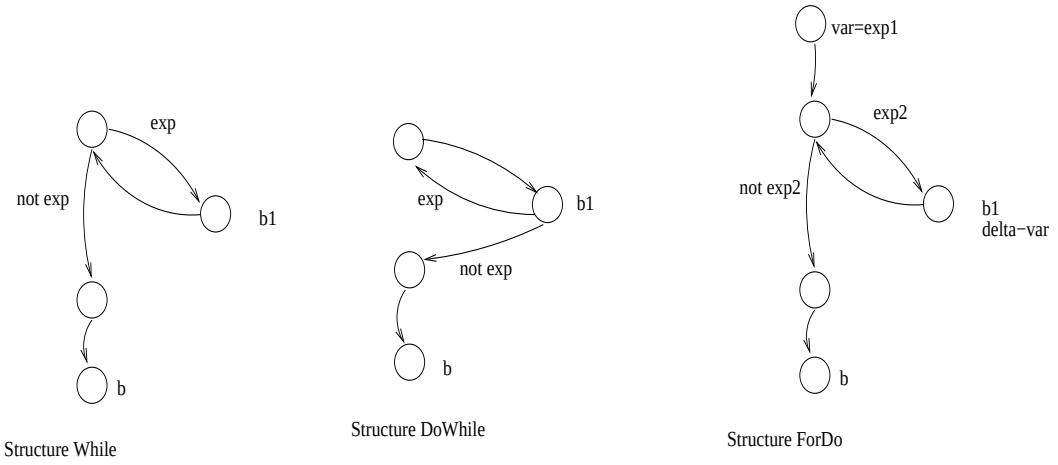

FIG. 4.4 – Représentation des structures répétitives du langage C

La méthode classique pour obtenir les blocs de base d'un programme consiste à en identifier les têtes et à découper le programme suivant ces têtes.

DÉFINITION – 4.2.1

*La **tête d'un bloc de base** (i.e. sa première instruction) est soit :*
- *la première instruction du programme,*
- *soit une instruction cible de branchement,*
- *soit l'instruction suivant un branchement.*

Ainsi, chaque bloc de base débute avec son instruction de tête et se poursuit jusqu'à la première instruction qui précède une autre instruction de tête ou un branchement.

Une représentation courante est d'annoter les arcs par la condition de branchement ou par la négation de cette condition. Notons que pour une fonction `f`, les conditions de branchements sont exprimées en fonction de ses variables d'entrée X mais aussi en fonction des variables locales à la fonction, que nous représenterons par le vecteur X_L avec p composantes pour une fonction utilisant p variables locales. Nous avons aussi besoin d'une variable supplémentaire contenant l'éventuelle valeur de retour de la fonction nommée `f_return`.

Les blocs de bases clairement identifiés, nous pouvons formaliser la construction du CFG d'une fonction.

DÉFINITION – 4.2.2

*Un graphe de flot de contrôle d'un programme ou **CFG** est le graphe connexe orienté étiqueté $G =< N, E, e, s, \delta, X, X_L >$ avec un unique nœud d'entrée e et un unique nœud de sortie s dont les nœuds sont les blocs de base du programme et tel que (b_i, b_j) est un arc du graphe si et seulement si l'une des deux conditions suivantes est vérifiée :*
- *la dernière instruction de b_i est un branchement à la première instruction de b_j ,*
- *b_j suit immédiatement b_i dans l'ordre du programme (ce qui implique b_j est un nœud successeur à b_i dans le CFG).*

X (resp. X_L) est le vecteur des variables d'entrée (resp. locales) de la fonction associé à ce CFG. La fonction d'étiquetage est définie comme :

$$\delta : (\delta_N, \delta_E)$$

$$\delta_N : N \rightarrow L_N$$

$$\delta_E : E \rightarrow L_E$$

avec L_N (resp. L_E) l'ensemble des blocs de base C associés à l'ensemble N (resp. l'ensemble des conditions associées à E) avec L_E et L_N les labels associés exprimés en fonction des variables de X et de X_L.

A chaque fonction correspond un ensemble fini de blocs de base et de branchements. Ainsi, en respectant la définition précédente, à chaque code source d'une fonction correspond un unique CFG.

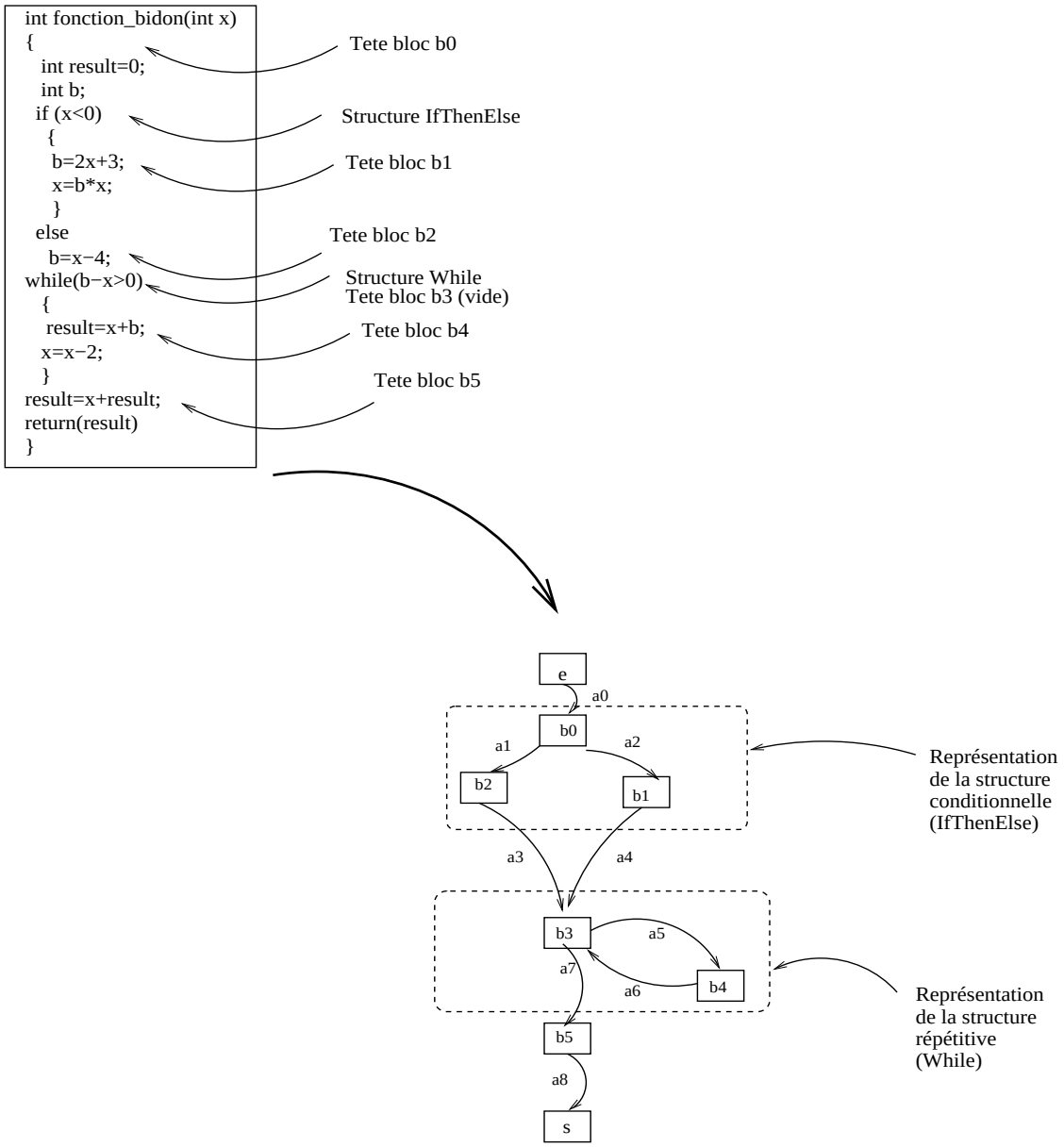

FIG. 4.5 – Construction de graphe de contrôle de la fonction `fonction_bidon`

Illustration 31
La figure 4.5 donne le code source de la fonction `fonction_bidon` *et le résultat de la construction de son CFG. Nous avons choisi, pour alléger la figure, d'annoter les arcs du CFG par a_i. Par exemple, l'arc noté a_0 correspond à l'arc $(e, b0)$.*

La fonction `fonction_bidon` *ne fait rien de particulier mais permet d'illustrer une structure conditionnelle et une structure répétitive ainsi que la mise en évidence des différents blocs de base de la fonction par identification de leurs têtes. Énumérons la construction en détails :*

- $\delta_N(b0) = \delta_N(b3) = \emptyset$
- $\delta_N(b1) = \{b = 2 * x + 3; x = b * x;\}$
- $\delta_N(b2) = \{b = x - 4;\}$
- $\delta_N(b4) = \{result = x + b; x = x - 2;\}$
- $\delta_N(b5) = \{result = x + result; return(result);\}$
- $\delta_E(a0) = \delta_E(a3) = \delta_E(a4) = \delta_E(a6) = \delta_E(a7) = \delta_E(a8) = \emptyset$
- $\delta_E(a1) = \{!(x < 0)\}$
- $\delta_E(a2) = \{(x < 0)\}$
- $\delta_E(a5) = \{(b - x > 0)\}$
- $\delta_E(a7) = \{!(b - x > 0)\}$

Étant donné le nombre d'instructions possible d'une fonction, le graphe peut facilement devenir d'une taille considérable et d'une grande complexité.

4.3 Chemins d'exécution et prédicats de chemin

Nous allons maintenant définir la notion de chemin d'exécution dans un CFG.

Définition – 4.3.1

Un ***chemin d'exécution total*** *d'un CFG* $G =< N, E, e, s, \delta, X, X_L >$ *correspond à un chemin total selon la définition B.2.1.*

Notation 11

La notation d'une séquence de chemin sous la forme $(ai, b, aj)^*$ *indique une séquence associée à une structure répétitive du code pouvant être itérée de 0 à un nombre infini ou donné de fois selon le type de la structure répétitive associée.*

En cas de présence d'une structure répétitive dans une fonction, chaque exécution de la fonction avec un nombre d'itérations différent dans la boucle induit un nouveau chemin d'exécution.

Notation 12

La notation Ch^* *représente l'ensemble des chemins dont la seule différence est le nombre d'itérations d'une séquence interne annotée* $(ai, bi, \ldots, bj, aj)^*$ *associée à une structure répétitive.*

Pour obtenir la trace symbolique d'exécution associée à un chemin total donné, nous appliquons la fonction d'étiquetage sur chaque élément de la séquence constituant ce chemin afin de récupérer la liste ordonnée des instructions exécutées et des conditions évaluées.

Définition – 4.3.2

La ***trace symbolique d'exécution*** *d'un chemin* $Ch(e, s) = (e, (e, b_0), \ldots, s)$ *notée* $\tau(Ch(e, s), \delta)$ *avec* δ *la fonction d'étiquetage du CFG associé vérifie :*

$$\tau(Ch(e, s), \delta) = (\delta_N(e), \delta_E((e, b_0)), \ldots, \delta_N(s))$$

avec $\delta_N(e) = \delta_N(s) = \emptyset$

Remarque(s) 15

Il pourra nous arriver par la suite de confondre le chemin d'exécution d'un CFG en terme de séquences d'arcs et de nœuds avec la trace symbolique d'exécution associée à ce chemin en terme d'instructions et de conditions du langage C exprimées en fonction des variables du chemin.

Les variables présentes dans la trace symbolique d'un chemin constituent un sous-ensemble des variables d'entrée, X, de la fonction ainsi qu'un sous-ensemble des variables locales de celle-ci, X_L : une variable d'entrée ou locale d'une fonction peut, en effet, ne pas être utilisée dans tous les chemins de la fonction. Cela nous amène à définir les variables d'un chemin $Var(Ch, G)$.

Définition – 4.3.3

> *Les **variables d'un chemin** $Var(Ch, G)$ avec $G =< N, E, e, s, \delta, X, X_L >$ le CFG correspond à l'ensemble des variables de X et de X_L utilisées et/ ou définies dans le chemin Ch tel que $Var(Ch, G) \subseteq X \cup X_L$.*

Pour notre stratégie de gestion des appels de fonctions, nous allons adapter la définition B.2.2 de chemin partiel d'un graphe de l'annexe B. Cela s'explique du fait que toute instruction d'appel dans le corps de la fonction sous test est contenu dans un chemin d'exécution total et que le sous-chemin allant de l'entrée e du CFG à l'instruction précédant une instruction d'appel permet de caractériser l'appel donné (nous verrons cela plus tard dans la partie III). La catégorie des sous-chemins d'exécutions partiels nous intéressant est donc les chemins partiels dont le nœud d'origine est l'unique nœud d'entrée e du graphe.

Définition – 4.3.4

> *Un **chemin d'exécution partiel** du CFG, $G =< N, E, e, s, \delta, X, X_L >$ est un chemin partiel $Ch(n_i, n_j)$ au sens de la définition B.2.2 avec $n_i = e$ et n_j tel que $(n_j, n_{j+1}) \in E$ avec n_{j+1} un nœud du graphe correspondant à une instruction d'appel définie dans la section 2.6.3.*

Remarque(s) 16

Par la suite, si nous ne précisons pas qu'un chemin est partiel, il s'agira par défaut d'un chemin d'exécution total. De façon générique nous noterons $Ch(e, n)$ un chemin partiel avec la notation $Chp(e, n)$.

Illustration 32

Reprenons la figure 4.5, un exemple de chemin d'exécution total :
$Ch(e, s) = (e, a_0, b_0, a_1, b_2, a_3, b_3, a_7, b_5, a_8, s)$ alors un exemple de chemin partiel peut être $Chp(e, b_0) = (e, a0, b0, a1)$ si $b2$ était un bloc de base contenant une instruction d'appel.

A un chemin d'exécution correspond un sous-domaine en entrée de la fonction entraînant l'activation de ce chemin dans le CFG de la fonction concernée. Pour définir le sous-domaine en entrée associé, il s'agit d'exprimer le prédicat de chemin i.e. les conditions à satisfaire sur les variables d'entrée de la fonction garantissant l'exécution du chemin associé.

Définition – 4.3.5

> *Soit un chemin d'exécution noté Ch extrait du CFG issu de l'implantation de la fonction f, le **prédicat de chemin** associé $PC(Ch, f, X)$ est la conjonction de contraintes exprimées sur X entraînant l'exécution de ce chemin. Un prédicat de chemin est une formule sur $\mathcal{L}$ universellement quantifiée sur X.*

L'algorithme de la méthode classique de calcul d'un prédicat de chemin est contenu dans la figure 4.6.

Nous désignerons par valeurs initiales des variables d'entrée comme les valeurs injectées à la fonction au moment de son appel c'est-à-dire les valeurs des variables d'entrée de la fonction avant toute définition.

Illustration 33

Imaginons cette fois que la trace symbolique d'exécution récupérée est

$$\tau(Ch(e, s), \delta) = ((result = 0), (x \geq 0), (b = x - 4), (b - x > 0), (result = x + result))$$

DEBUT
Pour un chemin Ch de $G =< N, E, e, s, \delta, X, X_L >$ calculer $\tau(Ch, \delta)$ et $Var(Ch, G)$
Pour chaque variable `var` dans $Var(Ch, G)$ faire
    $i \leftarrow 0$,
    annoter par i chaque occurrence de `var` avant toute définition
    Pour chaque définition de `var` par une expression `exp` faire
        $i \leftarrow i + 1$
        Tant que `var` n'est pas redéfinie faire
            indicer `var` par i
        Fin Tant que
    Fin Pour
    Pour chaque utilisation de $\texttt{var}_i$
        Si $i \neq 0$ et $var \notin Var(Ch, G)_{|X}$
            remplacer $\texttt{var}_i$ par `exp` jusqu'à une nouvelle définition de `var`
        Fin Si
    Fin Pour
Fin Pour
$PC(Ch, f, X) \leftarrow$ conjonction des conditions de $\tau(Ch, \delta)$ ainsi modifié
FIN

FIG. 4.6 – Algorithme classique de calcul de prédicat de chemin

Appliquons la première étape de l'algorithme sur les variables du chemin de la fonction (indicer à zéro les variables avant définition) :

$$(result_0 = 0), (x_0 \geq 0), (b_0 = x_0 - 4), (b_0 - x_0 > 0), (result = x_0 + result_0)$$

Pour la seconde étape, on incrémente les indices des variables à chaque définition :

$$(result_0 = 0), (x_0 \geq 0), (b_0 = x_0 - 4), (b_0 - x_0 > 0), (result_1 = x_0 + result_0)$$

La dernière étape consiste à exprimer les variables du chemin par leur expression exprimée en fonction des valeurs initiales d'entrée

$$(result_0 = 0), (x_0 \geq 0), (b_0 = x_0 - 4), (x_0 - 4 - x_0 \geq 0), (result_1 = x_0 + 0)$$

Le prédicat associé correspond à la conjonction des conditions C du chemin exprimées en fonction des valeurs initiales de variables d'entrée soit :
$PC(Ch, fonction\_bidon, x) = (x_0 \geq 0) \wedge (x_0 - 4 - x_0 \geq 0) = (x_0 \geq 0) \wedge (-4 \geq 0)$

Un **chemin d'exécution est dit faisable** quand le système de contraintes associé à son prédicat est satisfiable c'est-à-dire possède une solution. Dans le cas contraire, on dit que le **chemin est infaisable** (ou non exécutable).

Illustration 34
Prenons l'exemple du chemin $Ch(e, s) = (e, a_0, b_0, a_1, b_2, a_3, b_3, a_7, b_5, a_8, s)$ *de la figure 4.5 donnant le CFG de la fonction* `fonction_bidon`*. La trace symbolique d'exécution correspondant à ce chemin est :*

$$\tau(Ch(e, s), \delta) = ((result = 0), (x \geq 0), (b = x - 4), (b - x \leq 0), (result = x + result))$$

ce qui correspond au prédicat de chemin composé du système de contraintes :

$$(x_0 \geq 0) \wedge (-4 \leq 0)$$

Ce système de contraintes possède plusieurs solutions dont : $(x_0 = 0)$, le chemin associé est donc faisable.

Prenons maintenant le prédicat de chemin de l'illustration 33. Le système de contraintes associé est :

$$(x_0 \geq 0) \wedge (-4 > 0)$$

Ce système est insatisfiable donc le chemin associé est infaisable.

Le domaine d'un chemin correspond à l'ensemble des valeurs en entrée de la fonction vérifiant les propriétés définies dans le prédicat de chemin associé.

Définition – 4.3.6

Soit un chemin d'exécution noté Ch de la fonction f, le ***domaine de chemin*** *associé vérifie :*

$$Dom(Ch) = \{(a_1 \dots a_n) | \mathcal{M} \models_\nu PC(Ch, f, X)\}$$

avec $Dom(Ch) \subseteq Def(f)$ et ν interprétation unique sur $\mathcal{M}$ telle que $\forall i, 1 \leq i \leq n, \nu(x_i) = a_i$ pour $X = (x_1, \dots, x_n)$.

Les différentes instructions d'un chemin permettent de définir des contraintes sur les variables en entrée de la fonction mais aussi des contraintes induites par le calcul du chemin sur les variables en sortie.

Un chemin d'exécution Ch peut, en effet, être représenté par une relation nommée le **calcul du chemin** notée $Ch(X, Y, f)$ entre un sous-ensemble des valeurs en entrée et un sous-ensemble des valeurs en sortie, sous réserve que les affectations présentes dans le chemin Ch soient exécutées.

Le codomaine d'un chemin Ch de la fonction f est tel que :

$$CoDom(Ch) = \{(b_1 \dots b_m) | \exists (a_1 \dots a_n) \in Dom(Ch), \mathcal{M} \models_\nu Ch(X, Y, f)\}$$

avec $Ch(X, Y, f)$ le calcul du chemin et $Y = (y_1, \dots, y_m)$.

4.4 Modification de la construction du CFG d'une fonction contenant des instructions d'appel

La définition usuelle d'une fonction imbriquée est une fonction déclarée et définie localement dans le corps d'une autre fonction. Si nous faisons le parallèle avec la déclaration d'une variable, nous pouvons dire qu'une fonction imbriquée est locale à la fonction contenant sa définition. Ce type de fonctions n'existe pas en langage C.

Convention 3
Nous allons cependant utiliser la notion de fonction imbriquée pour désigner une fonction appelée en langage C.

Nous avons fait ce choix pour deux raisons.

- considérant que, par la suite, nous allons distinguer régulièrement les fonctions C selon qu'elles soient appelantes ou appelées, nous trouvons que le discours sera moins ambigu si nous parlons de fonctions imbriquées et appelantes plutôt que de fonctions appelantes et appelées.
- comme nous nous intéressons aux calculs faits par les fonctions, nous pouvons voir le calcul d'une fonction appelée comme imbriqué dans celui de l'appelante : en effet, le calcul de l'appelant précédant l'appel est exécuté puis celui de la fonction imbriquée et enfin le calcul de l'appelant suivant l'instruction d'appel.

Convention 4
Dans la suite de ce document, nous désignerons de façon générique par `f` *la fonction appelante et par* `g` *la fonction imbriquée. Certains exemples feront exception mais nous le préciserons dans ce cas.*

4.4.1 Caractérisation d'une fonction imbriquée en langage C

Une **fonction imbriquée** respectivement de façon directe ou indirecte dans une fonction `f` est une fonction respectivement utilisée dans le corps de `f` ou utilisée dans le corps d'une fonction elle-même utilisée dans le corps de `f`.

Ainsi pour chaque fonction imbriquée dans `f`, il existe au moins une exécution de `f` entraînant l'exécution de cette fonction imbriquée (sauf si l'instruction d'appel est contenue dans un chemin infaisable).

Reprenons la précédente définition, si une fonction `f` utilise directement une fonction récursive `g` alors la fonction `g` est imbriquée directement dans `f` et directement dans elle-même. Par voie de conséquence, la fonction `g` est aussi imbriquée indirectement dans `f` : son premier appel est contenu dans le code source de `f` et les suivants dans son propre code source.

Maintenant que nous avons introduit la notion de fonction imbriquée et d'appel de fonction, nous allons modifier légèrement la construction d'un graphe de flot de contrôle (cf. section 4.2). Ce point demande en particulier de redéfinir la notion de bloc de base.

4.4.2 Isolation d'une instruction d'appel dans un bloc de base

Un **bloc de base** est soit un bloc de base au sens de la définition 4.1.1 soit une instruction d'appel.

Ainsi, toute instruction C correspondant à une instruction d'appel de fonction constituera maintenant un bloc de base à part entière.

Cela implique aussi la modification de la définition de la tête d'un bloc de base.

Définition – 4.4.1
> *La* ***tête d'un bloc de base*** *est soit une tête de bloc de base au sens de la définition 4.2.1 soit une instruction d'appel ou l'instruction suivant une instruction d'appel.*

A ces définitions près, la méthode classique pour obtenir les blocs de base d'un programme et la définition du CFG donnée dans la section 4.2 restent identiques.

4.4.3 Illustration

Illustration 35
Nous allons nous appuyer sur le code source de la fonction f de profil $f : int \times int \rightarrow int$ *dont le code C correspondant contient un appel de la fonction g de profil* $g : int \times int \rightarrow int$. *Le code source de la fonction f est présenté dans la figure 4.7 et le graphe de contrôle associé est dans la figure 4.8.*

```
1   int   f(int x1,int   x2)
2   {
3     x1=  x2+x1;
4     if  (x1>10)
5        x1=(3*x1);
6     else
7        x2=x1;
8     x1=g(x1,x2);/*bloc de base d'appel*/
9     return(x1); /*bloc de base*/
10  }
```

FIG. 4.7 – Code source de la fonction f

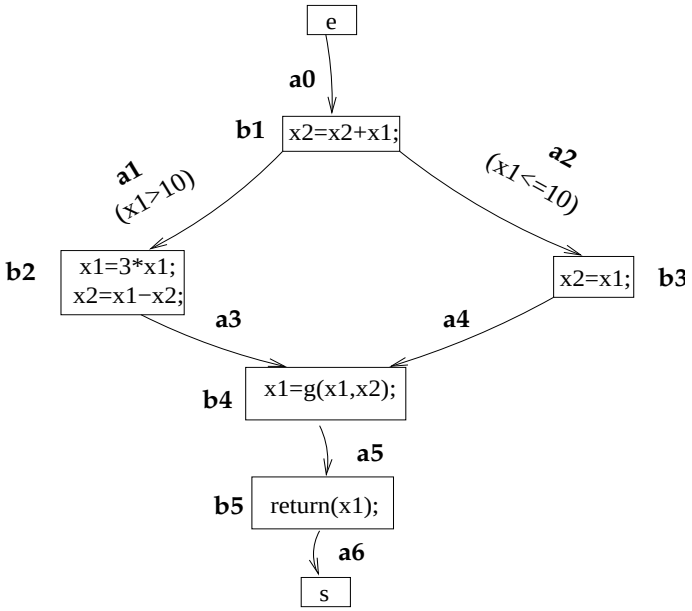

FIG. 4.8 – Graphe de contrôle de la fonction `f`

Le dernier point à aborder pour terminer la description du contexte de cette thèse concerne le thème du test de logiciel. Le chapitre suivant a pour objectif de rappeler la notion de test que ce soit unitaire ou d'intégration, fonctionnel ou structurel mais essentiellement d'expliciter une terminologie qui sera réemployée dans ce manuscrit.

Chapitre 5

Introduction au test de logiciel

Ce chapitre va nous permettre d'introduire les notions et définitions nécessaires sur le test de logiciel. Nous nous intéresserons ici au vocabulaire de base du test ainsi qu'aux techniques abordées dans ce manuscrit à savoir le test unitaire, le test d'intégration et les techniques structurelles et fonctionnelles.

Les techniques de test de logiciel sont très nombreuses et variées. Nous ne tentons pas de dresser un panorama exhaustif mais de nous intéresser aux techniques pertinentes par rapport à notre travail.

5.1 Définitions

Dans cette section et la section suivante, nous allons définir plus ou moins formellement de multiples notions du test de logiciels. Ces sections peuvent être ignorées par un lecteur familier au test de logiciel. En cas de doute sur une terminologie, le lecteur pourra toujours se référer à l'index des concepts et définition en page 12 pour consulter la définition associée. Nous nous bornons uniquement aux notions et à la terminologie utiles pour la suite. Pour l'ensemble de ces définitions, nous nous appuyons sur l'ouvrage de G. Myers [Mye79].

5.1.1 Test de logiciel

La définition du test selon l'I.E.E.E STD (paru en décembre 1990) est la suivante :

"Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond aux spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus "

L'AFCIQ (Agence Française de Contrôle Industriel de la Qualité) donne, quant à elle, la définition suivante :

« Technique de contrôle consistant à s'assurer, au moyen de l'exécution d'un programme que son comportement est conforme à des données préétablies »

et selon G. Myers [Mye79] dans son livre intitulé The Art of Software Testing

« Tester c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts. ».

Ces définitions permettent de mettre en avant l'objectif du test à savoir la conformité à la spécification (expression des fonctionnalités et du comportement désirés en langage formel, semi-formel ou naturel) du programme.
Le principal objectif de l'activité de **test logiciel** est d'examiner ou exécuter un logiciel dans le but d'y trouver des défauts.

L'origine, la cause d'une erreur lors de l'exécution d'un programme, est désignée comme un **défaut** ou plus communément comme un **bogue** (ou "bug").

Les techniques de test se déroulent en quatre étapes principales :

1. la sélection des cas de test basée sur l'analyse du code source et/ou de la spécification de la fonction sous test,
2. l'exécution (ou la simulation de l'exécution) de la fonction pour les valeurs en entrée précédemment déterminées (implique la mise en place d'une technique d'observation du comportement de la fonction et/ou de ses sorties),
3. le verdict de test qui consiste à déterminer si les sorties obtenues à l'étape précédente sont conformes à la spécification (problème de l'oracle) et
4. l'arrêt du test quand l'objectif est atteint que ce soit en termes de qualité de test ou de couverture de test.

5.1.2 Autres définitions nécessaires

Cette section peut apparaître comme un catalogue des concepts de test logiciel. Nous en sommes conscients mais il est nécessaire de caractériser toutes les notions utiles pour la suite et également de préciser la terminologie utilisée dans ce manuscrit. Nous rappelons que nous nous basons essentiellement sur [Mye79].

Un **cas de test** correspond au couple des valeurs des vecteurs d'entrée injectées à la fonction et de sortie obtenues après exécution de la fonction sous test (si celle-ci termine).

Remarque(s) 17
Nous utiliserons aussi le terme de cas de test pour désigner les valeurs en entrée déterminées pour une exécution donnée du programme sous test.

Un **critère de test** détermine les objectifs de couverture des cas de test :
- soit un ensemble précis de points ou portions du code source de la fonction (pour une détermination dite structurelle),
- soit un ensemble précis de fonctionnalités du logiciel définies dans la spécification de la fonction (pour une détermination dite fonctionnelle).

Définition – 5.1.1

Soit P une représentation d'un programme sous test et C le critère de test appliqué. Soit $E_C(P)$ l'ensemble des éléments de P caractérisé par C et soit $E'_C(P)$ l'ensemble des éléments de P caractérisé par C couverts lors des différents cas de test. La ***couverture de test*** *$Couv$ d'une fonction correspond au rapport suivant :*

$$Couv = \frac{E'_C(P)}{E_C(P)}$$

Définition – 5.1.2

Soit P une représentation d'un programme sous test et C le critère de couverture. Soit $E_C(P)$ l'ensemble des éléments de P caractérisé par C. Un critère C est couvert par une méthode de test sur P si tous les éléments de $E_C(P)$ ont une probabilité $q_{C,N}(D)$ d'être exécuté pour n cas de test avec la probabilité $q_{C,N}(D)$ représentant la ***qualité de test*** *associée.*

Un **jeu de test** est défini comme l'ensemble des cas de test successifs obtenus par une technique de test sur un logiciel donné.

Un **objectif de test** est un comportement ou un point critique du code que l'on cherche à exécuter, à atteindre.

Définition – 5.1.3

Soit $P : Def(P) \mapsto CoDom(P)$ le programme sous test, $S : Def(P) \mapsto CoDom(P)$ sa spécification, un ***oracle de test ou verdict de test*** *est la fonction de profil*

$$\mathcal{O} : CoDom(P) \times CoDom(P) \rightarrow \{true, false\}$$

telle que :

$$\forall X_i \in Def(X), \mathcal{O}(P(X), S(X)) = true \text{ si et seulement si } P(X) = S(X)$$

avec X_i le vecteur des valeurs des variables d'entrée.

Un **test exhaustif** est la technique de test optimale mais non réalisable en réalité : cette technique consiste à soumettre au programme tous les vecteurs de valeurs d'entrée possibles.

Ferguson et Korel distinguent deux types d'approches quant aux méthodes de test structurel [FK96] :
- les méthodes dites orientées but et
- les méthodes dites orientées chemin.

Une méthode de test **orientée chemin** est une méthode de test cherchant à déterminer des données de test en vue de couvrir un chemin (structurel ou fonctionnel pour un automate par exemple) donné de la fonction.

Une méthode de test **orientée but** est une méthode de test cherchant à déterminer des données de test en vue de couvrir un point donné de la fonction et ce quelque soit le chemin amenant à atteindre ce point.

Pour une méthode orientée chemin, un chemin est sélectionné dans le graphe de flot de contrôle atteignant l'objet cible. En revanche, pour une méthode orientée but, l'étape de sélection du chemin atteignant l'objet cible est éliminée.

L'**hypothèse d'uniformité** consiste à admettre que toutes les variables d'un sous-domaine en entrée de la fonction vont donner un même verdict de test. Cela peut être vérifiée par exemple par des techniques statiques d'analyse de code.

L'**instrumentation** consiste en l'ajout d' instructions de trace dans le code afin de connaître les parties du code activées par un cas de test.

Un **chemin manquant** caractérise un chemin d'exécution absent de l'implantation d'une fonction correspondant à une fonctionnalité attendue de la spécification et par conséquent non implantée.

Une partie d'implantation d'un programme qui n'est en réalité jamais exécutée est nommée du **code mort**.

5.2 Cycle en V et niveaux de test

5.2.1 Cycle en V

Le cycle de vie d'un logiciel appelé cycle en V (cf. figure 5.1) regroupe la totalité des opérations de développement d'un logiciel depuis l'analyse du besoin exprimé dans la spécification jusqu'à la réception du client.

La conception, au sens de la norme ISO9001, quant à elle, représente pour nous l'ensemble des activités de la branche descendante du cycle en V. La branche montante du V représente les différents contrôles et essais.

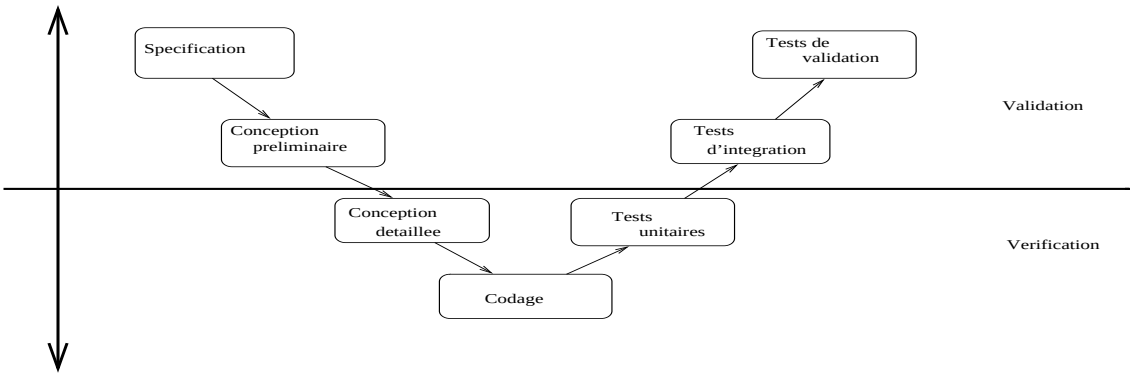

FIG. 5.1 – Cycle de vie d'un logiciel ou cycle en V

Le cycle en V est séparé en deux niveaux : les activités de vérification et les activités de validation.

5.2.2 Validation et vérification

La **vérification** a pour but de démontrer que les produits logiciels issus d'une phase du cycle de développement sont bien construits.

De façon générale, les activités de vérification répondent aux préoccupations du développeur :

" Ai-je fait UN bon logiciel ?"

Il s'agit de savoir si le logiciel a bien été conçu d'un point de vue implantation. La vérification se situe au niveau du concepteur du logiciel. Elle a également pour but de détecter et de rendre compte des fautes qui peuvent avoir été introduites au cours des phases précédant la vérification. L'objet de la vérification est donc de s'assurer de la bonne fabrication des différents éléments et de leur non défaillance afin d'atteindre un niveau de confiance satisfaisant.
Les activités de validation répondent aux préoccupations de l'utilisateur :

"Est-ce LE bon logiciel ?".

Il s'agit de vérifier si le logiciel répond bien aux attentes de l'utilisateur.

La **validation** correspond au processus d'évaluation du logiciel à la fin de son développement pour s'assurer de l'absence de défaillances et en particulier de la correspondance aux spécifications des besoins. La validation se situe au niveau des attentes de l'utilisateur. Les activités de validation s'effectuent à travers l'utilisation de différentes techniques de tests ou de preuve qui assurent la correction du programme.

5.2.3 Différents niveaux d'activités de test

Nous voyons que des activités de test se situent à différents niveaux sur le cycle en V du logiciel (cf. figure 5.1).

Test unitaire

Un **composant** représente le plus petit programme comportant des spécifications propres d'un logiciel.

L'activité de test qui consiste donc à tester indépendamment les différents éléments qui composent un logiciel est nommée le **test unitaire**.

La problématique associée au test unitaire est la suivante :

"Est-ce que mon composant logiciel est bien implanté ?"

Les premiers tests soumis au logiciel ont pour cible les composants élémentaires de l'application à tester c'est-à-dire chaque élément qui compose le logiciel est testé individuellement. Le test unitaire fait partie des activités de vérification et concerne la conception.

Test d'intégration

L'activité de test qui consiste à tester l'agencement des différents éléments qui composent un logiciel est nommée le **test d'intégration**.

La problématique associée au test d'intégration est la suivante :

"Est-ce que mes composants logiciel fonctionnent correctement ensemble ?"

Un test d'intégration est un test qui se déroule dans le cycle en V après les tests unitaires. Une fois tous les composants testés unitairement , ceux-ci sont regroupés et soumis à un test global. L'intégration a pour but de valider le fait que toutes les parties développées indépendamment fonctionnent bien ensemble. Le test d'intégration concerne les activités de validation comme d'ailleurs le test de validation.

Test de validation

L'activité de test qui consiste à tester l'ensemble d'un logiciel en vue de valider ses fonctionnalités est nommée le **test de validation**. On parle également de test système.

La problématique abordée est cette fois :

"Est-ce que mon logiciel fait bien ce qu'il devrait faire ?"

Le test de validation permet de vérifier si toutes les exigences client décrites dans le document de spécification d'un logiciel, écrit à partir de la spécification des besoins, sont bien respectées.

Cette activité de test termine le cycle en V du logiciel.

5.3 Différentes techniques de test

Il existe différentes classes de méthodes de test de logiciel selon le fait que le programme soit exécuté ou non pendant le test. Deux de ces méthodes sont clairement différenciées.

5.3.1 Test statique

La technique de test statique introduit fin 70 [Fag], [Mye78] repose sur l'analyse d'un logiciel sans exécution de son code. Il s'agit d'effectuer des analyses statiques détaillées du code source ou des spécifications du logiciel à partir des différents scénarii (les cas de test).

Le test statique peut servir à détecter le code mort et par conséquent les chemins du graphe infaisables (ou non exécutables) qui en découlent. Cette méthode n'entraîne aucune exécution du logiciel, tout cas de test ainsi que son résultat étant déduit de l'analyse. Cette méthode permet d'optimiser l'écriture du code source en procédant à une factorisation des éventuels bloc d'instructions répétés et donc de limiter son nombre d'erreurs.

5.3.2 Test dynamique

Le terme de test est utilisé parfois pour désigner l'action d'exécuter le logiciel avec des données d'entrée choisies afin de vérifier le comportement par analyse des données de sorties. Ce principe de test, le plus intuitif pour tout programmeur, désigne en réalité une sous-classe de test : le test dynamique.

Nous pouvons remarquer dans les précédentes définitions du test données par l'AFCIQ et par G. Myers que le terme de test désigne plus précisément le test dynamique car ces deux définitions suggèrent une exécution du composant.

Il s'agit de la technique de test par excellence [FK96], [GBR00], [WMMR05]. Cette méthode s'appuie sur des exécutions du programme pour déterminer progressivement un cas de test qui atteint un point donné du logiciel. Ce type de test inclut :

- la sélection des données de test,
- l'ajout éventuel d'instructions de trace pour suivre le chemin exact d'exécution,
- l'exécution du logiciel sous test
- et l'analyse des résultats.

Il apparaît aussi évident que, puisque le test est directement lié à l'exécution du programme, la validité de cette activité est strictement dépendante des conditions d'environnement du test. Ces conditions doivent refléter au plus près les conditions normales d'utilisation, autrement dit l'environnement opérationnel. Les analyseurs dynamiques de couverture structurelle permettent de mesurer le pourcentage d'objets testés grâce à l'instrumentation. Ils sont associés à un analyseur statique qui permet de générer les graphes d'appel (représentation graphique des interactions entre les différents composants d'un logiciel) pour les tests de validation et les graphes de flot de contrôle ou de flot de données pour les tests unitaires.

5.3.3 Exécution et évaluation symbolique

L'exécution et l'évaluation se trouvent à la frontière des méthodes de test statiques et dynamiques. L'**exécution symbolique** est une technique qui construit un prédicat de chemin et des expressions symboliques déterminant les valeurs symboliques de sortie en fonction des valeurs symboliques d'entrée pour un chemin donné. Les valeurs des paramètres formels sont remplacés par des valeurs symboliques et les instructions rencontrées le long du chemin sont exécutées selon leur sémantique.

L'**exécution symbolique dynamique** est une technique qui exécute un programme instrumenté. On obtient ainsi par des données d'entrée réelles, les données réelles de sortie, le chemin emprunté et l'expression symbolique du calcul de chemin associé.

La **méthode d'évaluation symbolique globale** construit la description complète de la sortie de la fonction en fonction des valeurs symboliques de ses données d'entrée pour la totalité de la fonction en se basant sur son graphe de contrôle. L'**évaluation symbolique globale** n'est pas exclusivement d'une méthode de test, elle est également utilisée pour des techniques de preuves

dont nous ne parlerons pas ici. L'évaluation symbolique a été introduite par James C. King en 1976 [Kin76]. Elle a fait l'objet de nombreuses études au début des années 80 mais peu d'outils performants existent réellement [Cow91], [CR85] car cette technique se prête difficilement à une utilisation industrielle. Elle permet de soumettre des tests plus élaborés que le test dynamique comme la soumission de tests génériques avec variables. Cette méthode n'est pas classée de façon stricte et englobe différents types d'exécutions symboliques.

Illustration 36
Soit la fonction suivante calculant le double du maximum de deux nombres :

```
int double_max (int a, int b)
{
   int x;
   if (a>b)
      x=a;
   else
      x=b;
   x=2*x;
   return(x);
}
```

L'évaluation symbolique globale de la fonction caractérise deux comportements :

$$((a > b, x = 2 * a), (a \leq b, x = 2 * b))$$

Remarque(s) 18
Pour l'évaluation symbolique globale, la complexité de mise en place de la technique est d'autant plus grande que la totalité du programme est évaluée en une seule fois contrairement à l'exécution symbolique qui va évaluer la fonction sous test chemin par chemin.

L'évaluation symbolique est utilisée selon trois optiques :
- le debogage et la compréhension d'un programme par étude du système récupéré,
- l'optimisation, la simplification ou la spécialisation d'une fonction par factorisation et formalisation du système,
- et l'application à la spécification formelle.

Son ambiguïté entre méthode statique et dynamique s'explique de par son fonctionnement. En effet, le programme est « exécuté » symboliquement par le testeur d'après la sémantique du code source. En d'autres termes, l'analyse statique du code source réalisée par le testeur (ou par le logiciel de test) permet de simuler l'exécution réelle du programme en lui injectant des variables d'entrées symboliques.

A cause de l'importance de l'analyse de la sémantique, certains testeurs classent cette méthode dans les méthodes statiques alors que d'autres préfèrent distinguer l'évaluation symbolique des autres méthodes statiques du fait de la simulation d'exécution.

Les méthodes de test sont aussi distinguées selon le critère choisi pour orienter les choix des données de test. Le choix de ce critère dépend tout d'abord de la représentation disponible du logiciel mais aussi du type de défauts que l'on cherche à mettre en évidence. Par exemple, les tests reposant sur le code source du programme ont une très bonne détection des défauts calculatoires du logiciel (division par zéro par exemple).

5.4 Techniques d'analyse

L'analyse permet de faire ressortir les caractéristiques principales d'un programme ou de sa spécification. Deux techniques sont distinguées :

– les techniques à critère fonctionnel reposant sur la description des fonctionnalités attendues du programme, en d'autres termes sur sa spécification,
– les techniques à critère structurel reposant sur l'analyse du code source du programme, de son implantation.

5.4.1 Critère fonctionnel

Une **technique de test fonctionnel** est une technique de test dont la détermination des différentes données de test repose sur une étude des spécifications de la fonction sous test.

Les techniques fonctionnelles sont depuis un certain temps privilégiées pour la génération automatique de cas de test comme pour [ABC+02], [OB88], [DF93], [MA00], ...

En effet, le test fonctionnel permet, contrairement au test structurel qui reste très concret, de choisir un niveau d'abstraction de la fonction à travers le niveau d'abstraction de la spécification.

Dans ce type de techniques de test, l'aspect implantation n'est pas analysé pour déterminer les données de test, seul l'aspect spécification du programme l'est. Le programme est testé à travers ses interfaces et la sélection des jeux de test va s'effectuer à partir des documents de spécification et de conception. L'idée est de vérifier le comportement réel du logiciel par rapport à son comportement spécifié. Les entrées, les sorties et leur relation sont étudiées. Le principe est de s'appuyer sur la spécification pour définir une partition sur l'espace des entrées définies par la spécification. Il s'agit de chercher la partition la plus fine possible de cet espace c'est-à-dire le plus petit sous-domaine comportant l'ensemble des valeurs des variables d'entrée provoquant un comportement donné. Pour chaque partition, une valeur d'entrée ainsi que la sortie correspondante (connue ou caractérisée d'après les spécifications) peuvent être sélectionnées par hasard ou sélectionnées plus ou moins près des bords du domaine afin de vérifier le respect de la spécification.

Une technique usuelle est de s'intéresser aux valeurs limites de chaque sous-domaine selon l'hypothèse que ces valeurs sont fortement pathogènes [LW90], [LPU02], [BDL06].

Les méthodes fonctionnelles évitent les problèmes d'analyse provoqués par le test structurel. Les tests fonctionnels semblent être les tests les plus près de la démarche naturelle des programmeurs.

Illustration 37
Un exemple est celui d'un développeur qui vient d'écrire une fonction calculant la valeur absolue d'un entier. Le premier test intuitif est d'exécuter la dite fonction avec une valeur en entrée négative puis avec une valeur en entrée positive et de vérifier la validité du résultat. En effet, la spécification du programme permet de déduire le comportement attendu du programme en fonction du variables d'entrée (ici la valeur absolue d'un entier) et l'exécution permet d'en vérifier la correspondance.

Le défaut des tests fonctionnels est le manque de justification de la représentativité des différents cas de test c'est-à-dire l'absence de l'hypothèse d'uniformité. Si toutes les données d'un logiciel ne peuvent pas être testées, seules certaines données dans les différents domaines sont testées. Dans un tel cas, l'hypothèse d'uniformité est nécessaire pour considérer les tests comme suffisants. De plus, le test fonctionnel ne permet pas toujours de mettre en avant toutes les erreurs issues de l'implantation, ce à quoi peut répondre le test structurel.

5.4.2 Critère structurel

Une **technique de test structurel** est une technique de test dont la détermination des différents cas de test repose sur une étude de la structure interne de la fonction sous test i.e. sur son implantation.

Les techniques de test structurel, quant à elles, ont pour objectif d'atteindre une couverture complète de certains objets du code source de la fonction sous test.

Test orienté flot de contrôle

Pour le test orienté flot de contrôle, il s'agit de couvrir les nœuds, les arcs et les chemins du graphe de contrôle représentant respectivement les instructions, les branchements ou les chemins d'exécution du code source [GBR00], [GDGM01], [Meu01], [WMM04a].

Illustration 38
En prenant l'exemple de la fonction suivante :

```
1.  void fonc(int x)
2.  {
3.    if (x<0)
4.       x=1-x ;
5.    if (x !=1)
6.       x=2*x;
7.  }
```

Pour le critère des instructions, il s'agit de tester au moins une fois toutes les instructions simples (pas de saut, de conditionnelle, ...) soit de couvrir les lignes 4 « `x=1-x ;` » et 6 « `x=2*x ;` ». Un test est suffisant avec la valeur -3 par exemple en entrée.

Pour le critère des branchements, il s'agit pour les expressions des instructions conditionnelles « `(x<0)` » et « `(x !=1)` » de les exercer vérifiées et non vérifiées. Deux tests suffisent avec les valeurs -3 et 1 en entrée.

Pour le critère des chemins, cela correspond aux cas où les deux conditions sont fausses, vraies, la première seulement est vraie et enfin seulement la seconde. Seuls trois de ces chemins sont exécutables. Le chemin correspondant à la vérification de « `(x<0)` » suivi après l'affectation « `x=1-x ;` » de la non vérification de « `(x !=1)` » est un chemin infaisable : en effet son prédicat de chemin correspond à un système de contraintes insatisfiable. Trois tests permettent de couvrir ces trois chemins avec en entrée -3, 1 et 4 par exemple.

Test orienté flot de données

Pour le test orienté flot de données, les critères (cf. [RW85]) s'intéressent à la couverture des relations entre les définitions et les utilisations d'une variable. Le graphe de flot de données correspond au graphe de contrôle de la fonction dont les nœuds sont annotés des définitions et utilisations des différentes variables de la fonction. La construction d'un jeu de test respectant ces critères n'est pas toujours possible et est indécidable dans le cas général. Les travaux de [HFGO94] ont pour but de comparer l'efficacité des critères flot de données et flot de contrôle. Selon ses expérimentations, l'efficacité des critères est équivalent et l'utilisation complémentaire des deux types de critères permet d'améliorer l'efficacité des tests. Notons que le critère orienté flot de contrôle tous-les-chemins reste le critère le plus rigoureux des critères structurels et que son application possède une efficacité au moins égale aux critères flots de données.

Test mutationnel

Le test mutationnel est une technique indirecte pour sélectionner ou évaluer une suite de tests pertinente. Cette technique a été introduite par DeMillo [DLS78] et par Offutt lors de ces travaux de thèse. Le score de mutation (nombre de mutants tués sur nombre total de mutants non-équivalents) est une mesure de qualité de la suite de test et donc de la qualité du logiciel. L'étude empirique de [ABL05] montre l'efficacité des suites de test générées par des techniques mutationnelles comme [BHJT00], [OAL06] ou encore [OVP]. La fiabilité de cette approche dépend du choix des opérateurs de mutation et de leur adéquation avec les erreurs réelles de la fonction sous test. Le nombre et la répartition des mutants générés ainsi que la présence de mutants équivalents (i.e. ayant le même comportement que la fonction sans erreur injectée) rend la méthode très coûteuse en temps d'exécution et de réalisation des tests. Le test mutationnel a donné lieu à de nombreuses publications que ce soit des études empiriques comme [Won93] ou pour la création de

nouveaux opérateurs de mutation. Cette technique reste principalement utilisée pour évaluer ou comparer des méthodes de sélection de cas de test.

Objectifs

Le but est de s'assurer que la totalité des différentes exécutions possibles (ou un nombre maximal) du logiciel a été explorée. L'avantage apparent des méthodes structurelles est de ne pas avoir besoin des spécifications. Cependant les spécifications restent nécessaires pour l'oracle. En effet, le comportement du composant à l'exécution peut être déduit après le test mais rien ne montre que ce comportement d'exécution est conforme au comportement exposé dans les spécifications. La conception d'un oracle n'est pas totalement écartée pour ce type de test, il suffit de le déduire par l'analyse des spécifications, si elles sont disponibles. De plus, une méthode structurelle ne garantit pas la détection des chemins manquants possibles [GG75] de la fonction.

Dans toutes les stratégies de test structurel étudiées, le but est de maximiser le rapport entre le nombre d'objets testés et le nombre des objets concernés par le critère de test choisi (c'est-à-dire de maximiser la couverture de test). Pour cela, différentes stratégies de choix des cas de test ont été mises en place.

Comme nous l'avons déjà dit, des critères structurels, le critère tous-les-chemins chemins demande le plus grand nombre de tests et possède donc également la plus grande chance, statistiquement, de détecter un bogue. De plus, l'application de ce critère rigoureux permet de mieux justifier l'hypothèse d'uniformité. En pratique, les techniques de test structurel, en particulier avec le critère des chemins, ne permettent d'atteindre que très rarement une couverture complète des objets à tester. Cela s'explique par la complexité et le nombre de calculs nécessaires pour déterminer les valeurs en entrée permettant d'exécuter chaque chemin.

5.5 Différentes stratégies pour déterminer les données de test

Un des meilleurs moyens pour atteindre un niveau de confiance satisfaisant serait de pouvoir sélectionner des cas de tests ayant le maximum de chances de détecter des défauts éventuels du produit logiciel sous test. Ce point est difficile et il faut aussi choisir le ou les tests selon des contraintes de coûts et de temps. La suite de ce chapitre explique différentes techniques de test proposées dans ce but et leurs caractéristiques.

Illustration 39

Étudions deux stratégies de test appliquées à une fonction calculant la somme de deux nombres :

- *une simple exécution du logiciel permet de vérifier si la sortie correspond au bon résultat pour un niveau de confiance bas (cas où la fonction n'est pas jugée importante),*
- *une exécution pour chaque combinaison possible d'entrées permet d'atteindre un niveau de confiance maximal (pour une fonction jugée comme primordiale). Ce type de test est exhaustif (toutes les entrées sont testées) mais n'est possible que lorsque les domaines des valeurs d'entrées ont un nombre fini d'éléments de taille raisonnable. Pour le cas de deux entrées entières le nombre de combinaisons s'élève à* $2^{32} * 2^{32}$.

Différentes stratégies de choix de cas de test peuvent aussi être distinguées selon la règle de sélection utilisée sur les valeurs des variables d'entrée. Du choix aléatoire des valeurs à l'élaboration de règles statistiques en passant par l'utilisation de règles combinatoires, la stratégie de sélection des cas de test peut-être très variée et dépend le plus souvent des domaines des valeurs des variables d'entrées et du graphe de contrôle du programme. Aucune stratégie n'est considérée comme la plus fiable dans tous les cas, chacune possède ses avantages et inconvénients, tout réside dans la justification du choix. Le problème principal étant de trouver la bonne combinaison entre le critère et la stratégie de test afin de détecter le maximum de défauts. Il est difficile de comparer sur ce plan test structurel et test fonctionnel par exemple. De plus, même si une technique nous paraît très efficace, il se peut que elle soit infaisable dans ce contexte de test

(données manquantes, méthode trop coûteuse,...) dans ce cas, on quantifiera la couverture des tests effectués.

5.5.1 Test exhaustif

Le test idéal est et restera le test exhaustif. Cependant celui-ci n'est pas toujours possible. Ainsi, si chaque combinaison donne un résultat correct et avec une hypothèse justifiable de déterminisme sur le programme sous test, la preuve de la correction du programme est faite. Il est facile de voir que le nombre de combinaisons possibles peut être très grand, le test exhaustif devenant alors long, fastidieux voire impossible dans certains cas.

L'idée est donc de limiter les cas de test à un nombre moindre selon un critère donné afin de maximiser la détection d'éventuels défauts. Les critères de sélection se justifient selon deux hypothèse : la classe d'équivalence ou l'hypothèse d'uniformité.

5.5.2 Stratégie aléatoire

Il s'agit d'une stratégie statistique répondant à une loi uniforme.

Comme son nom l'indique, le test aléatoire consiste à injecter « au hasard » différentes valeurs aux variables d'entrée du programme à étudier. Le composant est exécuté avec des données choisies aléatoirement dans le domaine de définition de la fonction. La quasi totalité des fonctions vont avoir des traitements différents selon les valeurs des variables d'entrées. La justification de cette technique réside sur la faible probabilité du programme de s'exécuter selon le même traitement tout au long de l'analyse ce qui correspondrait à une succession de tests avec des valeurs d'entrées similaires. Ainsi, plus le nombre de cas de test est grand, plus le pourcentage d'objets testés est considéré comme élevé [GDGM01]. Le problème principal étant qu'il faut un très grand nombre de tests pour espérer tester tous les traitements possibles et que certains d'entre eux seront testés plusieurs fois alors que le test d'autres traitements seront testés plus tard (si on arrive à les atteindre).

5.5.3 Stratégie statistique

Le principe du test statistique [GDGM01], [TFW91] est dérivé de la génération de cas de test aléatoire. Il s'agit de couvrir en priorité les éléments de probabilité minimale définis par un critère de test donné fonctionnel ou structurel. Des éléments de probabilité minimale sont les éléments ayant statistiquement le moins de chance d'être couverts. Dans [TFW91], la détermination des éléments de probabilité minimale est manuelle et dans [GDGM01], elle est automatique.
Il s'agit donc d'une stratégie aléatoire orientée.

5.5.4 Stratégie combinatoire

Le critère de test combinatoire ou «model-based» [CDFP97] s'applique aux programmes (ou aux spécifications) ayant des variables d'entrée qui prennent leurs valeurs dans des domaines discrets et assez petits. L'exemple typique est un programme où toutes les variables d'entrée sont de type booléen. Pour cet exemple, le test exhaustif est possible sauf pour un très grand nombre de variables d'entrée. La génération des cas de test est alors facilité par construction de toutes les combinaisons légitimes des valeurs d'entrée [GMSB96]. Le test combinatoire garantit la détection de toute erreur qui dépend d'une certaine combinaison de valeurs d'un sous-ensemble de 1 à n variables d'entrée du programme ou de sa spécification. Le test sera donc fait avec toutes les combinaisons possibles de valeurs de chaque sous-ensemble de n variables d'entrée.

5.5.5 Analyse

Remarque(s) 19
Plus la complexité d'un code est grande en termes de nombre de lignes et de nombre de structures conditionnelles et répétitives, plus la probabilité pour celui-ci de contenir des erreurs est élevée.

Atteindre un niveau de confiance satisfaisant ne signifie pas avoir établi formellement la correction du produit logiciel. En effet, si le test ne détecte aucune anomalie, rien ne prouve qu'un test plus élaboré n'en découvrira pas.

Nous reprenons la citation bien connue et acceptée dans la communauté du test de E. W. Dijkstra extrait de "Notes On Structured Programming" en 1970 :

"Program testing can be used to show the presence of bugs, but never to show their absence"

Il faut en particulier éviter une mauvaise utilisation de la stratégie de test qui consisterait à choisir des tests uniquement pour illustrer le bon fonctionnement du produit logiciel. De plus, pour des parties cruciales de certains logiciels critiques, il peut être judicieux de suivre des méthodes de preuve en complément des méthodes de tests [DGM93]. Les méthodes de test fonctionnelles et structurelles sont complémentaires : chacune couvre un type d'anomalies spécifiques et aucune ne peut donc être jugée comme étant la plus performante. Il est important de noter que, puisque le test n'est qu'un outil de validation partielle, il n'est pas question de dégager une préférence pour une méthode de test plutôt qu'une autre. Seule une étude et une bonne évaluation du problème orientent le choix de la technique de test.

Dans cette partie, nous avons posé les bases nécessaires à la lecture de ce manuscrit en définissant ou redéfinissant tous les concepts utiles.

Nous avons ainsi précisé toutes les notions jugées importantes en ce qui concerne le langage C, la modélisation proposée des fonctions par notre langage de spécification, la notion de graphe de flot de contrôle d'une fonction et pour finir les notions et la terminologie du test de logiciel utilisée dans ce manuscrit.

Toutes les briques principales ayant été posées, nous allons pouvoir aborder dans la partie suivante le premier thème principal de cette thèse à savoir le test unitaire structurel.

Deuxième partie

Génération de cas de test unitaires structurels

Cette partie est consacrée aux méthodes de test unitaire à critère structurel.

Ce type de techniques concerne la méthode de test unitaire PathCrawler présentée dans ce manuscrit.
En quelques mots, la méthode PathCrawler est une méthode de génération de cas de test unitaire dynamique et itérative appliquant le critère structurel des chemins et destinée aux langages impératifs séquentiels et en particulier au langage C pour lequel un prototype a été implanté.

Dans un premier chapitre, nous allons présenter le problème général de la génération automatique de cas de test structurel et dresser un panorama de quelques techniques de test choisies pour leur similitude avec PathCrawler ou pour illustrer certaines difficultés des méthodes structurelles.

Le chapitre suivant s'attardera sur la méthode PathCrawler, méthode qui a été initialement proposée dans le cadre de mon stage de D.E.A. [Mou03] et qui a été mûri et améliorée depuis. Nous y détaillerons son principe de base, ses principales caractéristiques ainsi que son fonctionnement détaillé.

Le chapitre suivant permettra d'illustrer cette méthode en déroulant son application sur une fonction C de tri fusion `merge` ce qui permettra de mettre ainsi en avant les points forts de la méthode.

Enfin, un dernier chapitre dressera un bilan de la méthode et permettra de justifier pourquoi nous avons choisi de remplacer le traitement "inlining" des fonctions appelées dans la fonction sous test par une nouvelle stratégie de gestion des appels de fonctions.

Chapitre 6

Problème ATDG

Dans ce chapitre, nous allons présenter d'une part la méthode classique utilisée pour la génération automatique de cas de test structurel (ATDG [1]) ainsi que les problèmes rencontrés lors de sa mise en place. Ensuite, nous dresserons un rapide panorama de quelques travaux existants choisis soit pour leur solution proposée aux problèmes d'ATDG soit pour leurs similitudes avec notre propre méthode de test.

6.1 Problème de la génération automatique de cas de test structurel

Commençons tout d'abord par présenter la méthode classique de génération automatique des cas de test pour l'application d'un critère structurel.

6.1.1 Méthode classique

La méthode courante pour la génération automatique des cas de test ou ATDG est composée de six étapes principales représentées et numérotées dans la figure 6.1.

1. Tout d'abord, via une analyse statique du code source de la fonction sous test, le graphe de flot de contrôle (CFG) est construit. Cette étape est représentée par la partie grisée de la figure 6.1.
2. A partir de ce CFG et du critère structurel appliqué, l'ensemble des objets du graphe à atteindre est déterminé.
3. On sélectionne tout d'abord un premier objet à couvrir. Il s'agit ensuite de sélectionner un chemin du CFG atteignant ce point (pour une méthode dite orientée chemin) ou l'ensemble des chemins atteignant ce point (pour une méthode dite orientée but).
4. A partir de ce ou ces chemins, on détermine alors le(s) prédicat(s) de chemin associé(s) vérifié(s) par toute donnée de test atteignant l'objet à couvrir si celui-ci est atteignable.
5. On recherche ensuite des données de test satisfaisant ce prédicat de chemin.
6. Dès qu'un objet du CFG est couvert, un nouvel objet est sélectionné parmi ceux restants à couvrir et les phases 3 à 5 sont répétées jusqu'au taux de couverture désiré.

Illustration 40
Prenons, comme exemple, la fonction C `getmid` *qui retourne la valeur intermédiaire de trois entiers passés en arguments dont le code se trouve dans la figure 6.2. Pour alléger notre exemple, nous ne réécrirons pas les conditions des structures conditionnelles : nous avons choisi de les annoter sous*

[1] Automatic Test Data Generation

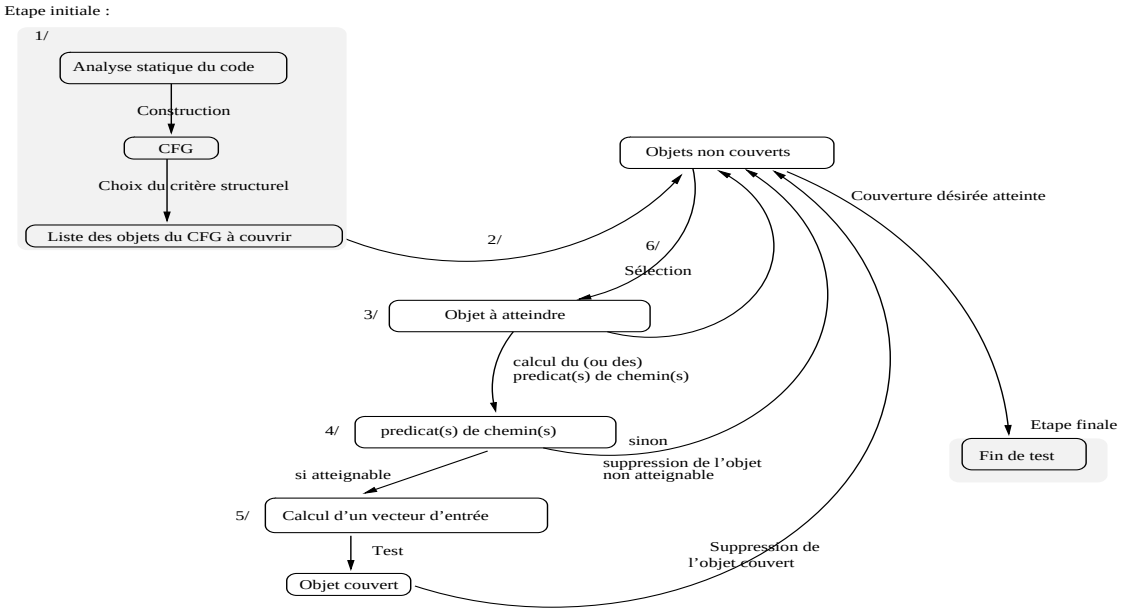

FIG. 6.1 – Déroulement de la méthode structurelle classique

la forme c_i (dont la correspondance est explicitée dans le code source de la fonction) et tel que $c_{i+1} = \neg c_i$ pour i impair $\in [1..11]$.

Tout d'abord, cette fonction est soumise à une analyse statique afin de récupérer et construire le CFG correspondant présenté dans la figure 6.3.

Si nous choisissons le critère des branchements, il faut tester chaque condition au moins une fois comme étant vérifiée et non vérifiée. Les 4 séquences de chemin suivantes :

1. $(e, b1, c1, b2, c3, b5, b7, b8, c8, b15, s)$,
2. $(e, b1, c1, b2, c4, b3, c6, b6, b7, b8, c7, b9, c9, b12, b14, b15, s)$,
3. $(e, b1, c1, b2, c4, b3, c5, b4, b6, b7, b8, c7, b9, c10, b10, c12, b13, b14, b15, s)$ *et*
4. $(e, b1, c2, b8, c7, b9, c10, b10, c11, b11, b13, b14, b15, s)$

permettent par exemple de couvrir la totalité des branchements.

Si nous choisissons un critère plus exigeant, celui des chemins, il faut tester toutes les exécutions possibles soit toutes les successions possibles des différents arcs du graphe, ce qui constitue au total 16 chemins dans le graphe soit 16 cas de tests différents.

L'exemple est plutôt simple ici car la fonction de la figure 6.2 ne comporte que 6 instructions conditionnelles. De plus, nous nous sommes limités uniquement à la détermination de l'ensemble des objets à couvrir (étape 2 de la figure 6.1). Cependant, il reste à déterminer les prédicats de chemin, de vérifier la satisfiabilité de ceux-ci pour, dans le cas positif, déterminer les données de test associées et dans le cas négatif (objet non atteignable) de recommencer ces étapes pour un autre objet à couvrir. Nous allons voir dans la section suivante la difficulté de mises en œuvre de ces étapes.

6.1.2 Les principaux problèmes

Pour des fonctions plus réalistes, les méthodes d'ATDG se heurtent à de nombreux problèmes que nous allons présenter en dépliant toutes les étapes de la méthode classique.

```
1  int getmid(int x1, int x2, int x3)
2    {  int mid;
3       mid = x3;
4       if (x2 < x3)        /* c1 = (x2<x3) et c2=(x2>=x3) */
5       {  if (x1 < x2)     /* c3 */
6            mid = x2;
7          else
8            if (x1 < x3)   /* c5 */
9              mid = x1;
10      }
11      if (x2>=x3)         /* c7 */
12      {  if (x1 > x2)     /* c9 */
13           mid = x2;
14         else
15           if (x1 > x3)   /* c11 */
16              mid = x1;
17      }
18      return mid;
19   }
```

FIG. 6.2 – Fonction `getmid`

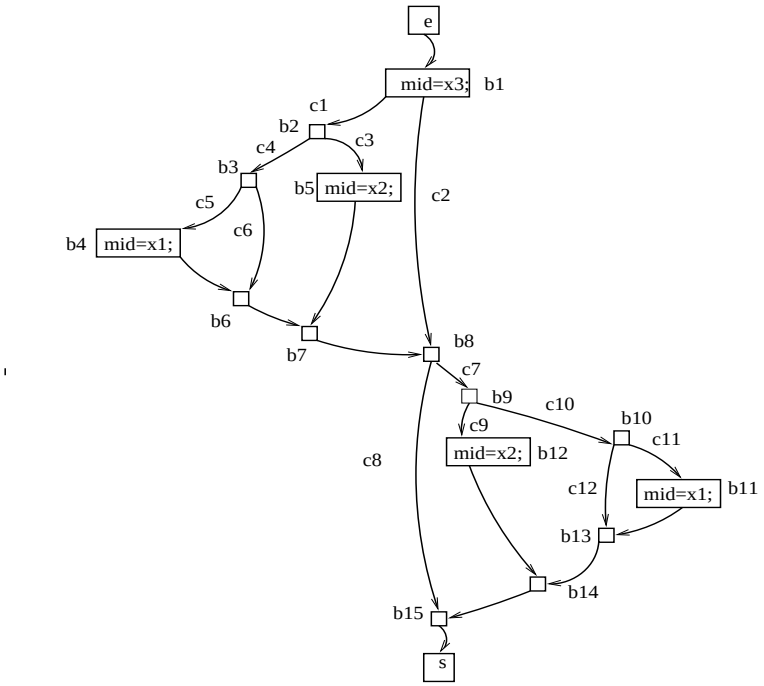

FIG. 6.3 – Graphe(`getmid`)

L'explosion combinatoire des chemins

Un des problèmes propres au test structurel apparaissant rapidement est le problème de la combinatoire des chemins. Ce problème n'est pas lié à la génération des cas de test mais concerne l'application de critère de test structurel. Cela s'explique facilement du fait qu'un critère de test fort exige un nombre élevé de cas de test.
Même avec une fonction simple comme la fonction `getmid`, nous nous apercevons que l'application

d'un critère structurel entraîne un nombre de tests élevé en particulier, pour le critère des chemins. En effet, ce critère demande un grand nombre de cas de test du fait que le nombre de chemins croît de façon exponentielle en terme des instructions conditionnelles ou répétitives présentes dans le code source de la fonction sous test. Pour cette raison, le critère des chemins, malgré le haut niveau de confiance qu'il procure, est en réalité peu appliqué pour le test et ce, même pour des systèmes jugés critiques.

Calcul des prédicats de chemins avec alias

Une fois un chemin ou un ensemble de chemins atteignant l'objet à couvrir identifié, il s'agit de déterminer les données de test correspondantes. Pour cela, une phase nécessaire est la détermination du (ou des) prédicat(s) de chemin associé(s).

Pour le calcul du prédicat de chemin à partir du chemin cible, il faut substituer les variables du chemin par leur valeur symbolique c'est-à-dire par leur expression en fonction des valeurs initiales des variables d'entrée. Il faut alors pouvoir identifier la valeur symbolique d'une variable à chaque instant de l'exécution, valeur qui est directement liée au chemin d'exécution suivi. La difficulté survient avec la gestion des données structurées et/ou des pointeurs, en particulier avec la gestion des variables synonymes ou alias.

Illustration 41
Prenons la section de code suivante :

```
1. int tab[10];
2. int *pt;
3. pt=&tab[1]+2;
4. *(pt+1)=5;
5. if (tab[4]==5)
6.  /* .... */
```

A la ligne 3, nous avons la création de deux chemins d'accès équivalents : `&tab[1]+2` *et* `pt` *qui représentent aussi le chemin d'accès* `&tab+3`*. L'instruction de la ligne 4 est équivalente à l'instruction* `*(&tab+3+1)=5;` *ou encore à l'affectation* `tab[4]=5;`*. Ainsi, l'expression de la ligne 5 est toujours vérifiée.*

La méthode simple de substitution de calcul de prédicat de chemin comme expliquée dans la figure 4.6 du chapitre 4 va associer à ce chemin la contrainte $PC = (tab[4] = 5)$ *ce qui signifie que la correspondance entre* `*(pt+1)` *et* `tab[4]` *n'est pas identifiée.*

La méthode simple de substitution de calcul de prédicat est insuffisante en présence de pointeurs. Deux optiques sont alors envisageables.

On peut utiliser des techniques de substitution plus élaborées c'est-à-dire prenant en compte à chaque instant du code une modélisation exacte de la mémoire pour permettre la prise en compte efficace et exacte des alias comme pour notre méthode de test expliquée dans la section 7.3.1. On peut aussi procéder à un renommage des variables de la fonction à chaque nouvelle définition pour la mise en place d'une mise sous forme statique à assignation unique comme dans [GBR00] par exemple. Il s'agit là, de la stratégie la plus souvent adoptée que nous verrons plus en détails un peu plus tard dans la section 6.2.1.

Détection des chemins infaisables

Une fois le (ou les) prédicat(s) de chemin déterminé(s), il s'agit de résoudre le(s) système(s) de contraintes associé(s) à ces prédicats dont toute solution constitue une donnée de test atteignant l'objet à couvrir.

Le problème concerne alors la résolution de ces systèmes de contraintes et en particulier la détection de leur insatisfiabilité. Pour montrer qu'un chemin est infaisable, il faut montrer

l'insatisfiabilité de la conjonction des contraintes de son prédicat. Ce problème est indécidable dans le cas général et dans le meilleur des cas, NP-complet pour des contraintes sur des variables prenant leurs valeurs dans des domaines discrets [JM94] c'est-à-dire pour le langage C, tous les domaines de types entiers C.

Nous rappelons que si un prédicat de chemin correspond à un système de contraintes insatisfiable alors le chemin qu'il modélise est dit infaisable.

Illustration 42
Reprenons l'exemple précédent 40 avec la fonction `getmid` *de la figure 6.2. Avec les mêmes séquences de chemins choisies pour couvrir le critère des branchements, nous obtenons les prédicats suivants :*

1. $(x2 < x3) \wedge (x1 < x2) \wedge (x2 < x3)$,
2. $(x2 < x3) \wedge (x1 \geq x2) \wedge (x1 \geq x3) \wedge (x2 \geq x3) \wedge (x1 > x2)$,
3. $(x2 < x3) \wedge (x1 \geq x2) \wedge (x1 < x3) \wedge (x2 \geq x3) \wedge (x1 \leq x2) \wedge (x1 > x3)$ *et*
4. $(x2 \geq x3) \wedge (x2 \geq x3) \wedge (x1 \leq x2) \wedge (x1 \geq x3)$.

L'insatisfiabilité des systèmes de contraintes est immédiate pour les prédicats 2 et 3 dans la mesure où ils contiennent deux contraintes contradictoires. Les deux autres prédicats (1 et 4) correspondent quant à eux à des systèmes de contraintes satisfiables ce qui signifie que des données d'entrée peuvent être déterminées pour couvrir ces deux chemins.

L'exemple est simple ici mais généralement, devant la complexité des expressions algébriques manipulées, la détection de chemins infaisables peut facilement rendre la méthode inutilisable.

Traitement des structures répétitives

Nous pouvons donner comme autre exemple d'instance de chemin infaisable le cas où le nombre maximal d'itérations pour une structure répétitive est indéterminable. Pour ce cas, le problème des chemins infaisables est lié à un autre problème : celui du traitement des structures répétitives comme la structure While présentée dans le chapitre 2 dans la section 2.3.4 et les structures répétitives ForDo et DoWhile présentée en annexe A dans la section A.2. Nous en distinguons deux classes :

- celles à nombre fixe d'itérations et
- celles à nombre variable d'itérations.

Une **structure répétitive à nombre fixe d'itérations** est une structure répétitive à nombre constant de passages dans le corps de la structure.

Une **structure répétitive à nombre d'itérations variable** est une structure répétitive dont le nombre de passages dépend des valeurs des variables de la fonction.

Il est également possible de rencontrer une structure répétitive infinie quand le nombre de passages est infini dans le corps de la structure répétitive (exemple d'une structure répétitive `while(1)`) c'est-à-dire en cas de non terminaison. De plus en présence de variables dynamiques, nous entendons toutes les variables du code qui sont allouées ou libérées dynamiquement, le problème de gestion des alias augmente par la possible création d'une infinité d'alias en cas de présence d'une allocation dynamique d'une variable de type pointeur dans une structure répétitive.

En ce qui concerne les structures répétitives avec l'application du critère structurel des chemins, différentes optiques sont adoptées :

1. Une première optique, la moins rigoureuse, consiste à couvrir la fonction avec un cas de test correspondant à aucune itération dans la structure répétitive et un cas de test correspondant à au moins une itération dans la structure répétitive. Cette stratégie est assez simple à mettre

en place mais procure un niveau de confiance trop faible dans la mesure où un seul cas de test entraînant au moins une itération de la structure répétitive ne peut garantir que cette structure est valide pour l'ensemble de ses itérations.

2. Une seconde optique, la plus rigoureuse, consiste quant à elle à couvrir toutes les itérations possibles de la structure répétitive. Le problème étant alors la difficulté d'application d'une telle stratégie en présence d'une fonction contenant de multiples structures répétitives (le nombre de cas de test à effectuer croit exponentiellement) et la possible présence de structures répétitives infinies.
3. Une autre optique, consiste à couvrir uniquement les chemins de longueur bornée, on parle du critère des chemins-long$_n$[2] avec n la longueur totale maximale des chemins couverts [Gou04]. La difficulté étant ici de déterminer la longueur n suffisante. On se rend compte que le nombre d'itérations couvertes dans la boucle dépend de la complexité du programme avec l'hypothèse que plus un programme est complexe, plus ses chemins d'exécution sont d'une longueur élevée. Cette limitation de la longueur totale des chemins couverts borne indirectement le nombre d'itérations des structures répétitives des chemins.
4. Enfin, la dernière optique, propose de tester toutes les itérations de la structure répétitive au nombre inférieur ou égal à k (le plus souvent fixé par l'utilisateur). Ce critère restreint donc la couverture aux chemins contenant au plus k itérations par structure répétitive y compris en présence de structures répétitives imbriquées. Cette dernière solution est celle adoptée fréquemment. Cette restriction du critère structurel des chemins se nomme le critère des **k-chemins** et possède un niveau de rigueur supérieur à la première optique présentée pour tout $k > 1$ et peut-être appliquée à des codes réalistes en adaptant la valeur de k. Rien ne garantit cependant qu'une erreur ne soit pas présente à partir de la $k + 1^{eme}$ itération de la structure répétitive. Ainsi, pour appliquer ce critère, il faut pouvoir justifier que ce nombre de passages est faisable et suffisant pour détecter les éventuelles erreurs du programme, si elles existent ce que l'on désigner par l'hypothèse de régularité.

6.2 Travaux de recherche antérieurs

Dans cette section, nous allons commencer par présenter quelques travaux répondant aux problèmes de l'ATDG précédemment présentés et en particulier au problème traitant de l'analyse syntaxique du code des fonctions.

Ensuite, nous allons dresser un rapide panorama non exhaustif de techniques existantes d'ATDG. Nous avons choisi de présenter différentes méthodes d'ATDG pour leurs similitudes avec notre propre technique de génération de cas de test structurel unitaire.

6.2.1 Forme SSA

Un moyen rencontré pour simplifier la détermination des prédicats de chemin (c'est-à-dire la traduction des instructions du chemin en terme de contraintes exprimées sur les valeurs initiales des variables d'entrée de la fonction sous test) est une mise sous forme statique à assignation unique (SSA) du composant sous test [CFK+91] comme dans [GBR00],[SD01].

[2] S. D. Gouraud nomme ce critère le critère des n-chemins mais pour éviter toute ambiguïté avec le critère des k-chemins, nous avons décidé d'utiliser une autre terminologie.

Définition – 6.2.1

> *La **forme SSA** est une version équivalente, sémantiquement parlant, d'une fonction où chaque variable possède une définition unique et dans laquelle chaque utilisation d'une variable fait référence à une définition unique.*

La mise sous en forme SSA consiste à renommer une variable dès que celle-ci est redéfinie. Ainsi, plus aucune ambiguïté ne repose sur les valeurs des variables. Les instances de variables peuvent être vues comme des variables à valeur unique tout comme des variables logiques. Cette modélisation simplifie le calcul des dépendances lors de l'analyse du programme, assure l'unicité des définitions et permet la commutativité des instructions à partir du moment où les définitions de variables ne sont pas dépendantes du flot de contrôle de la fonction (c'est-à-dire hors présence de structure conditionnelle et/ou répétitive). Chaque variable étant renommée à chaque redéfinition, l'ordre des instructions devient sans importance, les instructions sont commutatives. Ainsi, les instructions sont vues comme des contraintes et sont indépendantes du chemin du graphe de contrôle.

```
1 int fonction (int x)
2 {
3     int a,b ;
4     a=12 ;/*définition de a*/
5     b=3*a;/*définition de b*/
6     b=b+2;/*définition de b*/
7     a=x*b;/*définition de a*/
8     return(a);
9 }
```

FIG. 6.4 – Fonction avant mise sous forme SSA

Illustration 43
Dans la figure 6.4, la variable `a` *prend successivement les valeurs 12 puis la valeur de l'expression* `x*b`*. De même, la variable* `b` *prend la valeur de l'expression* `3*a` *puis la valeur de l'expression* `b+2`*. La valeur des variables* `a, b` *dépend donc de la ligne d'instruction considérée dans le code. Après la mise sous forme SSA de la figure 6.5, plus aucune confusion n'est possible : la variable* `a1` *vaut 12,* `a2` *vaut la valeur de l'expression* `x0*b1`*,* `b1` *vaut l'expression* `3*a1` *et la variable* `b2` *vaut la valeur de l'expression* `b1+2`*.*

```
1 int fonction (int x0)
2 {
3     int a0,b0 ;
4     a1=12 ;
5     b1=3*a1;
6     b2=b1+2;
7     a2=x0*b1;
8     return(a2);
9 }
```

FIG. 6.5 – Fonction après mise sous forme SSA

Tout d'abord, le nom des valeurs initiales des variables sont indicées de la valeur zéro. Hors présence de structurelle conditionnelle et/ou répétitive, le code source est analysé séquentiellement et à chaque définition de variable, le nom de cette variable est indicé de l'indice de sa valeur courante plus un et à chaque utilisation de variable, le nom de celle-ci est indicé de l'indice de sa valeur courante. Par cette technique, la valeur d'une variable peut être connue pour un chemin donné

mais cela introduit de nombreuses variables supplémentaires qui ne simplifient pas le problème du traitement des alias.

L'unicité des définitions n'est pas aussi simple dès que le code source de la fonction contient des structures répétitives et/ou conditionnelles introduisant de nouveaux flots de contrôle.

```
1  void fonction (int x)
2  {
3      int a,b ;
4      a=12 ;
5      b=3a;
6      if (x>0)
7           a=x;
8      else
9           a=b;
10     a=2*a;
11 }
```

FIG. 6.6 – Autre fonction avant mise sous forme SSA

Illustration 44
Si nous regardons la figure 6.6, la valeur de la variable `a` *à la ligne 10 est dépendante de la vérification de la structure conditionnelle c'est-à-dire correspond soit à la valeur de l'expression symbolique* `2*x` *soit à* `2*3*12`.

Ainsi, lors de la présence de structures de contrôle, des assignations particulières sont mises en place par l'introduction de phi-fonctions [CFK+91] retournant l'ensemble des valeurs courantes possibles d'une variable à un point du code donné selon le flot de contrôle précédemment suivi. Ces fonctions sont utilisées quand une définition de variable et donc son renommage dépend du chemin suivi.

Illustration 45
Reprenons la figure 6.6, sa mise sous forme SSA avec utilisation des phi-fonctions est contenue dans la figure 6.7. La fonction spéciale `phi-fonction(a2,a3)` *utilisée dans la section 6.7 renvoie l'instance de la variable* `a` *dépendante de la vérification de la structure conditionnelle.*

```
1  void fonction (int x1)
2  {
3      int a,b ;
4      a1=12 ;
5      b1=3*a1;
6      if (x1>0)
7           a2=x1;
8      else
9           a3=b1;
10     a4=2*phi-fonction(a2,a3);
11 }
```

FIG. 6.7 – Seconde fonction après mise sous forme SSA

6.2.2 Méthode points-to analysis

Pour la gestion des pointeurs (cf. section 2.5.3), la méthode "points-to analysis" de Emami, Ghiya et Hendren [EGH94] est très souvent utilisée comme dans [GBR00]. Cette méthode calcule

un sur-ensemble de relations de pointages i.e. toutes les variables pouvant être pointées. La nature de chaque pointage est prise en compte ou sens "relation de pointage certaine" et "relation de pointage probable". En effet, selon le chemin d'exécution suivi, un pointeur peut faire référence à différentes variables si sa relation de pointage est définie dans une structure conditionnelle ou répétitive du langage C. Dans un tel cas, la relation de pointage est directement dépendante du flot de contrôle suivi lors de l'exécution de la fonction.

```
1  void fonction (int x,int y,int z)
2  {
3      int *p1;
4      int *p2;
5      int *p3;
6      p1=&x;
7      if (x>y)
8      {
9        p2=&y;
10       p3=&z; /*point A*/
11     }
12     else
13     {
14       p2=&z;
15       p3=&y; /*point B*/
16     }
17     p1=&z;     /*point C*/
18 }
```

FIG. 6.8 – Fonction exemple pour la méthode points-to

Les relations de pointage certaines sont notées (p, q, D) ce qui signifie que la variable `p` pointe sur la variable `q`. Les relations de pointage possibles sont notées quant à elles (p, q, P) c'est-à-dire que `p` peut pointer sur `q`. D dénote donc une relation de pointage certaine et P une relation de pointage probable.

Illustration 46
Étudions le cas de la figure 6.8 aux points A, B, C du code. Les relations de pointages sont liées au flot de contrôle de la fonction suivi à l'exécution. L'application de la méthode points-to aux différents points de la fonction nous donne :
- $A : (p1, x, D), (p2, y, D), (p3, z, D)$
- $B : (p1, x, D), (p2, z, D), (p3, y, D)$
- $C : (p1, z, D), (p2, y, P), (p3, z, P), (p2, z, P), (p3, y, P)$

Ainsi si un pointeur `p` pointe vers un entier, alors toutes les variables de type entier susceptibles d'être pointées, lui seront liées par la fonction spéciale *points_to*().

Illustration 47
Ainsi pour la fonction de la figure 6.8, au point C du code source, l'ensemble des relations de pointage pouvant être associées aux pointeurs sont :
- $p1 = points\_to(\&z)$
- $p2 = points\_to(\&y, \&z)$
- $p3 = points\_to(\&y, \&z)$

Nous remarquons que la fonction points_to possède un seul argument pour les relations de pointage certaines et plusieurs arguments pour les relations de pointage probables.

Dans le cas de relations de pointage non certaines, chaque scénario doit être pris en compte lors de l'analyse de la fonction sous test.

Nous allons maintenant présenter des travaux de recherche d'ATDG classés selon leur similitude avec notre approche. Nous retenons les caractéristiques principales de notre approche à savoir l'aspect dynamique, l'utilisation de la PLC pour la stratégie de sélection des cas de test et la détermination des prédicats de chemin, l'application du critère des chemins et l'aspect adaptatif de la méthode.

6.2.3 Aspect dynamique

Les méthodes dynamiques de génération de cas test structurel sont assez nombreuses comme par exemple [FK96], [KWF92], [GN97], [MM98], [GMS98]. Le grand avantage de ces méthodes concerne la gestion des données dynamiques et le problème des chemins infaisables qui est dans certains cas simplifié voire éliminé. En particulier, la génération dynamique aléatoire de cas de test, exécute la fonction sous test en générant au hasard des données de test jusqu'à atteindre la couverture de test désirée. Les chemins exécutés sont tous par définition faisables. Pour une génération aléatoire et sans analyse du code source, seuls les chemins exécutés sont analysés et par conséquent, les chemins infaisables ne sont donc pas identifiés ainsi que le code mort qui en découle. Le point faible des techniques dynamiques est l'utilisation d'heuristiques pour la recherche des données de test [Kor90], [MM98], [GN97].

Nous avons choisi de présenter une des approches dynamiques les plus connues à savoir la méthode de Bogdan Korel fonctionnant sur une heuristique de recherche de minima de fonction [Kor90].

Il s'agit d'une méthode dynamique de génération automatique de cas de test structurel respectant le critère "toutes les instructions", pour des programmes implantés en Pascal. Elle s'appuie sur une technique d'énumération des valeurs sans aucune propagation de contraintes. Elle s'apparente à une technique "générer et tester" (cf. annexe C) dans la mesure où le test de cohérence des valeurs se fait après les instanciations des variables du système de contraintes associé au problème de la génération du cas de test suivant.

La recherche de minima de la fonction de branche a pour objectif de définir un cas de test satisfaisant un chemin donné supposé faisable, le chemin Ch. Un premier cas de test, X_1, est sélectionné aléatoirement. Le chemin emprunté Ch_1 est récupéré lors de l'exécution. Si Ch_1 est identique à Ch alors l'objectif est atteint. Sinon, on détermine F comme la fonction de branche de la première condition de branchement différenciant Ch de Ch_1.

Pour atteindre le chemin cible à couvrir, il s'agit de minimiser des fonctions de branches modélisant la différence entre le dernier chemin couvert et le chemin cible.

Nous notons *op* les opérateurs d'égalité ou d'inégalité. Chaque contrainte du prédicat de chemin est donc de la forme $E1$ *op* $E2$ avec Ei des expressions arithmétiques exprimées sur les valeurs initiales des variables d'entrée. Ces contraintes sont transformées sous la forme F *rel* 0 où F est une fonction de branche comme définie par Korel valant $E2 - E1$ ou $E1 - E2$ et *rel* est la relation découlant de *op* avec $rel \in \{>, >=, =\}$.

Illustration 48
Ainsi pour l'expression `( x>=10 )`, *la transformation nous donne* $10 - x \geq 0$ *soit la fonction de branche* $F(x) = 10 - x$.

Si le prédicat de la branche est vérifié, la fonction de branche correspondante F est négative ou nulle et inversement. Cette approche évalue les fonctions des branches pour chaque valeur d'entrée lors de l'exécution.

On détermine le sens de recherche via la recherche de minima de la fonction de branche pour savoir si on doit incrémenter ou décrémenter les valeurs (pour plus de détails sur la recherche de

solutions consulter [Kor90]) et ainsi de suite jusqu'à détermination d'un vecteur d'entrée couvrant le chemin cible Ch.

Le problème ici est que, dans de nombreux cas, différents minima locaux peuvent être déterminés. Cette méthode peut échouer dans sa recherche de solutions quand la modification du vecteur d'entrée n'influe pas directement sur la fonction de branche évaluée tout simplement si la fonction de branche correspondante du chemin est inatteignable en réalité. Ces deux points en font une méthode peu applicable en réalité car la détermination des cas de test reste difficile dans la mesure où elle repose sur une heuristique de recherche.

6.2.4 Utilisation de la PLC

La puissance de la PLC[3] (cf. annexe C) en fait un outil de plus en plus présent dans les méthodes de test structurel [GDGM01], [GBR00], [SD01], [Meu01], [BCH+04] ou également pour des méthodes fonctionnelles [PO01], [MA00].

Nous avons choisi ici de présenter la méthode Inka [Got00], [GBW06], [GBR00] car il s'agit de la méthode de test structurel jugée la plus proche de notre approche au moment de sa création.

Cet outil a été réalisé dans le cadre de la thèse d'Arnaud Gotlieb [Got00] dirigée par Michel Rueher et Bernard Botella. L'outil permet de résoudre le problème de la génération automatique de cas de test structurel par la mise en œuvre d'une approche s'appuyant sur l'utilisation de la PLC [GBR00]. Cette approche s'adresse aux programmes implantés en langage C ou C++. Un prototype a été réalisé dans le cadre du projet RNTL Inka, il analyse le fichier source préprocessé et génère un système de contraintes associé. A partir du code source du programme sous test et d'un critère structurel, la fonction de l'outil Inka consiste à fournir un jeu de données d'entrée du programme vérifiant la couverture requise.

La première étape est la mise sous forme SSA (présentée dans la section 6.2.1) du composant sous test. Ce point est important pour deux raisons :
- la résolution de contraintes est implantée en Prolog,
- l'analyse syntaxique du code source est simplifiée par l'unicité des définitions de variables.

Le programme est transformé en un système de contraintes traduisant la globalité du programme. Ce résultat peut être comparé à une évaluation symbolique globale (cf. chapitre 5) où l'ensemble d'un programme est modélisé sous un ensemble d'expressions algébriques traduisant l'algorithme de celui-ci.

Le choix d'un point à atteindre dans le CFG permet de poser des contraintes supplémentaires devant être satisfaites pour atteindre le point sélectionné : il s'agit donc d'une méthode orientée but. Ces contraintes sont obtenues par analyse des dépendances dans le programme entre les variables utilisées dans le point sélectionné (instruction ou condition de branchement) et les définitions de ces mêmes variables. Le programme sous test et un point sélectionné du graphe sont ainsi tous deux transformés en un système de contraintes équivalent. Si le système de contraintes est inconsistant alors le point sélectionné ne peut être atteint. Une solution d'un tel système de contraintes constitue une donnée de test qui atteint le point sélectionné. La méthode diffère selon que la faisabilité d'un chemin est prise en compte ou pas. Si elle n'est pas prise en compte, seules les instructions précédentes au point sélectionné sont considérées ; sinon toutes les instructions sont considérées car un chemin peut être infaisable à cause d'une instruction successive au point sélectionné.

Dans certains cas, le temps alloué pour la résolution du système de contraintes est insuffisant, cela conduit à changer d'heuristique. Certains cas spécifiques restent toutefois inaccessibles par

[3] Programmation Logique avec Contraintes

l'indécidabilité du problème de la génération automatique de cas de test structurel en présence de structures répétitives dont la terminaison ne peut être garantie.

La résolution de ce système s'appuie sur les techniques classiques de la PLC (techniques de filtrage, énumération des valeurs, propagation de contraintes) comme expliquées en annexe C. Pour le choix des valeurs des variables d'entrées, la stratégie Inka consiste à diviser le domaine de chaque variable en deux pour tester la consistance aux bords des sous-domaines. La détection d'un sous-chemin infaisable permet d'éliminer les valeurs inconsistantes des domaines des variables et donc de réduire l'espace de recherche. Pour traiter la gestion des pointeurs, la méthode SSA est étendue en une forme Pointeur-SSA avec utilisation de la méthode points-to analysis.

La plupart des opérateurs du C sont traités y compris les opérateurs de traitements bit à bit et les opérateurs logiques. La récursivité est également traitée.

La nouvelle version Inkav2 permet la prise en compte des flottants, des structures dynamiques, des pointeurs, des données dynamiques mais les instructions non structurées (`goto, break, continue,...`) ainsi que les instructions spécifiques d'entrée/sortie, le transtypage (modification d'un type d'une variable) et les aspects polymorphiques du langage C++ restent écartés de l'analyse.

Notre approche est comparable en de nombreux points à cela mais diffère de cette méthode par la gestion de certaines instructions non structurées, le transtypage, par le critère structurel choisi plus rigoureux. De plus, notre stratégie de sélection des cas de test simplifie le problème des chemins infaisables. Cependant, nous n'avons pas encore inclus la dimension objet dans notre stratégie.

6.2.5 Critère structurel des chemins

Il n'existe pas de travaux antérieurs sur la génération des cas de test des chemins car le critère tous_les_chemins est quasiment inapplicable pour de nombreuses fonctions réelles. Cependant, des travaux de recherches proposent une restriction de ce critère comme dans [GDGM01] ou [SMA05] qui se limitent à la couverture des chemins-long$_n$ c'est-à-dire des chemins de longueur maximale n ou à une application du critère des k-chemins, k bornant le nombre d'itérations par structure répétitive comme dans notre approche [Mou04]. D'autres travaux comme [LY00], [MS04] ou comme [GMS98] sont des méthodes orientées chemin qui utilisent un algorithme génétique ou d'une méthode itérative basée sur une élimination gaussienne.

Cependant, comme nous l'avons dit précédemment, les méthodes de test orientées-chemin contiennent une phase de sélection des chemins supplémentaire par rapport à une méthode orientée but. Ce problème ajouté au nombre de tests induits par l'application du critère des chemins justifie la non application de ces méthodes à l'ensemble des chemins du CFG de la fonction sous test.

Nous avons choisi ici de présenter le travail de thèse de Sandrine-Dominique Gouraud [Gou04] concernant une méthode de génération de cas de test structurel statistique pour les critères toutes les instructions, tous les branchements ainsi que tous les chemins-long$_n$ pour du code C.

Comme nous l'avons déjà dit dans le chapitre 5, le test statistique est dérivé du test aléatoire ajouté de probabilités sur les objets du CFG pour les méthodes statistiques structurelles. Le test aléatoire correspond à une méthode statistique avec probabilité uniforme. Pour de plus amples informations sur les méthodes de test statistiques consultez [TFW91].

L'idée proposée dans [Gou04] est d'automatiser la proposition faite par [TFW91] qui est d'augmenter la qualité de test en couvrant en priorité les objets du graphe à probabilité minimale.

Pour cela, l'ensemble C_n des chemins de longueur inférieur ou égal à n est identifié avec c_n le nombre des chemins de cet ensemble. L'ensemble des chemins de C_n passant par un objet e du graphe est noté C_n^e.

La difficulté consiste à déterminer la bonne valeur de n assurant une bonne qualité de test : l'optique choisie est de fixer n comme la longueur maximale des chemins minimaux de la fonction. Un chemin minimal est identifié comme un chemin ne contenant que des circuits élémentaires ou nuls.

Pour une méthode de test aléatoire avec SA l'ensemble des objets du graphe à couvrir alors la probabilité de tirer un élément e de SA est de $p(e) = \frac{1}{|SA|}$. Cependant, la probabilité de couvrir $e \in SA$ ne correspond pas à la probabilité de tirer e directement dans la mesure où on peut couvrir e en tirant un autre élément de SA. Ainsi, la probabilité de couvrir $e \in SA$ avec c_k le nombre de chemins passant par k et c_k^e le nombre de chemins passant par k et couvrant également e se définit comme

$$p'(e) = \frac{1}{|SA|} + \frac{1}{|SA|} * \sum_{k \neq e} \frac{c_k^e}{c_k}$$

avec $\sum_{k \neq e} \frac{c_k^e}{c_k}$ la probabilité de couvrir e en tirant l'élément k.

Si les éléments e et k sont des chemins du graphe alors $\sum_{k \neq e} \frac{c_k^e}{c_k} = 0$. L'idée est de limiter le tirage à un sous-ensemble de SA pour limiter la cardinalité de l'ensemble où s'effectue le tirage et donc d'augmenter les probabilités minimales et par conséquent d'améliorer la qualité de test.

Pour l'application du critère des chemins-long$_n$, SA est réduit à un unique élément obligatoire de la fonction correspondant à un bloc ou à un arc obligatoire c'est-à-dire contenu dans tous les chemins de C_n.

Une fois un chemin tiré, le prédicat de chemin est défini et le système de contraintes associé est résolu par l'utilisation de la PLC. Toute solution de ce système constitue une donnée de test pour le cas de test suivant.

En présence de structures répétitives dans le code, le critère des instructions et des branchements est préféré au critère des chemins-long$_n$ de par le nombre de tests qu'il impose. De plus, la qualité de test obtenu pour ce critère est souvent moindre que pour les deux premiers comme l'explique S.-D. Gouraud dans [Gou04].

Cette méthode a donné de bons résultats sur des exemples académiques. Un prototype AuGuSTe a été implanté mais se limite actuellement aux fonctions et procédures C ne possédant que des passages d'arguments par valeurs et ne traitant qu'un sous-ensemble restreint du langage (pas de types de données complexes, d'instructions de modification de flot, ...).

6.2.6 Aspect adaptatif

Nous pouvons citer trois méthodes adaptatives de test structurel se basant sur la structure du CFG de la fonction sous test pour la stratégie de sélection des cas de test. La méthode la plus connue est celle de Bogdan Korel nommée "chaining-approach" [FK96] mais nous pouvons aussi citer [McC96] ou encore [PM87].

Nous ne présenterons pas en détail ici la méthode de Korel [FK96] fonctionnant sur une étude des dépendances des nœuds du graphes en termes de définition et d'utilisation de variables couplée à une fonction de minimisation pour déterminer les différentes données de test. En revanche, nous allons nous attarder sur la méthode de Prather et Myers [PM87] qui, certes, n'a jamais été implantée à notre connaissance mais qui, contrairement aux autres méthodes dynamiques, n'a pas l'objectif d'atteindre un point donné du CFG mais repose, comme pour notre méthode, sur une réutilisation d'un préfixe du prédicat de chemin du cas de test courant.

Cette méthode applique le critère des branchements. Une première exécution aléatoire du code permet de couvrir un premier chemin d'exécution de la fonction. A partir de ce chemin, le prédicat de chemin associé est exprimé sous la forme d'une conjonction ordonnée des prédicats de branches parcourues :

$$PC = c_1 \wedge \ldots \wedge c_n$$

avec c_i le prédicat de la i^{eme} branche du chemin déterminée par substitution arrière.

L'étape suivante consiste alors à déterminer tous les préfixes réversibles de ce prédicat de chemin.

Définition – 6.2.2

Pour un prédicat de chemin

$$PC = c_1 \wedge \ldots \wedge c_n$$

un ***préfixe réversible*** *est un préfixe de PC de la forme* $c_1 \wedge \ldots \wedge c_i$ *avec* $i \leq n$ *tel que le système* $c_1 \wedge \ldots \wedge \neg c_i$ *soit satisfiable.*

Cette notion de préfixe réversible sera reprise dans le chapitre 7 présentant notre méthode de test unitaire PathCrawler.

Le vecteur d'entrée du prochain cas de test correspondra alors à une solution du plus petit préfixe réversible identifié du prédicat de chemin courant. Les éventuels préfixes réversibles non sélectionnés sont stockés en mémoire.

Ce vecteur d'entrée permet de couvrir un nouveau chemin du CFG et donc de déterminer un nouveau prédicat de chemin sur lequel on applique la même stratégie. Si aucun préfixe réversible n'est identifié pour un prédicat de chemin, on réutilise un préfixe réversible précédemment identifié mais non encore utilisé qui permet d'atteindre une branche non couverte et ainsi de suite jusqu'à la couverture de tous les branchements de la fonction.

Illustration 49

Prenons l'exemple de la fonction :

```
void fonc(int x1, int x2)
{
 int r;
 if (x1==1) /*c1*/
  r=1;
 else
  r=2;
 if (x1+x2==1) /*c2*/
  r=r+2;
 else
  r=r+3;
 if (x1*x2==1) /*c3*/
  r=5;
}
```

Un premier vecteur d'entrée $X1 = (0, 0)$ *correspond à* $PC = \neg c1 \wedge \neg c2 \wedge \neg c3$. *Le plus petit préfixe réversible est* $\neg c1$ *ce qui nous donne pour le second cas de test* $X2 = (1, 0)$ *par exemple. Le préfixe réversible* $\neg c1 \wedge \neg c2$ *non utilisé est stocké (le préfixe* $\neg c1 \wedge \neg c2 \wedge \neg c3$ *est non réversible ici).*

Le prédicat de chemin couvert est ensuite $PC = c1 \wedge c2 \wedge \neg c3$. *Le plus petit préfixe réversible devient* $c1 \wedge c2$ *(*$\neg c1$ *ayant déjà été couvert et* $c1 \wedge c2 \wedge \neg c3$ *est non réversible).*

La branche restant à couvrir est ici c3, le vecteur d'entrée $X_3 = (1, 1)$ *permettrait d'atteindre cet objectif, son prédicat de chemin associé étant* $PC = c1 \wedge \neg c2 \wedge c3$. *Cependant, le seul préfixe réversible* $\neg c1 \wedge \neg c2$ *ne permet pas de couvrir cette branche.*

Ainsi, une branche accessible peut ne pas être atteinte par cette méthode de par le fait que rien ne garantit que les préfixes réversibles identifiés passent par toutes les branches du CFG.

6.2.7 CUTE : une méthode proche

Nous allons également nous attarder sur trois méthodes [GKS05], [CE05], [SMA05] qui présentent plusieurs similitudes avec notre approche, en particulier la méthode CUTE [SMA05] qui propose de couvrir tous les chemins-long$_n$ faisables d'une fonction C (ou Java). Cette méthode procède tout d'abord à une transformation du code source de la fonction sous test et une instrumentation mise en place via l'analyseur syntaxique CIL [Lan06]. Elle utilise une **"concolic"** exécution définie dans [SMA05] comme une exécution **conc**rète et symb**olique** de la fonction sous test :

- une première exécution concrète sur le code instrumenté permet de construire un modèle symbolique de la fonction et
- différentes exécutions symboliques de ce modèle permettent d'atteindre la couverture désirée.

Cette méthode propose donc de couvrir tous les chemins-long$_n$ faisables de la fonction sous test par une méthode adaptative basée sur un solveur de contraintes pour les contraintes linéaires et l'utilisation du backtracking et réutilisant la structure du prédicat de chemin courant pour mettre en place une exploration en profondeur d'abord bornée du CFG symbolique de la fonction sous test.

Les valeurs concrètes des variables de type pointeur sont approximées par des valeurs abstraites. Ces approximations peuvent amener à de possibles exécutions de chemins non faisables et de façon symétrique la non exécution de tous les chemins faisables de la fonction sous test. Notons ici que les similitudes avec notre méthode de test (cf. chapitre 7) sont très nombreuses mais que nous traitons les contraintes non-linéaires et que nous nous distinguons par notre traitement des alias que nous expliquerons dans le chapitre 7.

Nous venons de présenter le problème de l'ATDG et les difficultés rencontrées lors de sa mise en place ainsi que différents travaux existants. Dans le chapitre suivant, nous allons présenter notre méthode de test structurel unitaire à savoir la méthode PathCrawler.

Chapitre 7

La méthode PathCrawler

Nous allons présenter dans ce chapitre notre approche pour la génération automatique de cas de test structurel que nous avons appelé la méthode PathCrawler. A la fin de ce chapitre, nous appliquerons la méthode sur un exemple correspondant à un noyau du langage C. Un exemple plus complet sera traité dans le chapitre suivant.

Cette approche concerne la quasi totalité du code C ANSI. Un premier prétraitement du code source transforme le code source sous test sous une forme canonique et simplifiée sémantiquement équivalente. *Cette transformation du code source permet d'obtenir le sous-ensemble du langage C étudié dans le chapitre 2.* Il s'agit d'une phase très importante de la méthode dans la mesure où elle simplifie la phase d'analyse et la mise en place de la stratégie. De plus, lors de ce prétraitement, une instrumentation du code est mise en place, nous permettant de récupérer la trace symbolique des chemins exécutés.

Nous allons tout d'abord commencer par donner l'idée directrice de la méthode dans une première section et ensuite nous irons plus en détails quant au fonctionnement et la mise en place.

7.1 Idée directrice

7.1.1 Caractéristiques et principe de base

Notre approche diffère des autres méthodes d'ATDG en plusieurs points. Tout d'abord, il s'agit d'une méthode **à la volée** ce qui signifie qu'au lieu de construire le CFG au début, un autre graphe de la fonction sous test (c'est-à-dire l'arbre des chemins faisables d'exécution) est construit au fur et à mesure des différents cas de test.

Pour être plus précis, nous construisons l'arbre des chemins d'exécution faisables de la fonction sous test. Cette stratégie représente une économie de coût qui serait lié à la construction préalable du CFG de la fonction sous test et évite d'énumérer les chemins du CFG y compris les chemins infaisables.

Notation 13
Nous notons le vecteur d'entrée du i^{eme} cas de test X_i comme les valeurs du vecteur des variables d'entrée X de la fonction sous test.

Contrairement aux méthodes classiques d'ATDG, nous ne définissons pas au préalable un objet du graphe à couvrir pour le prochain cas de test, nous ne pouvons donc par parler d'approche orientée chemin ou orientée but. Notre stratégie est semblable à [PM87] c'est-à-dire que la détermination du prochain cas de test se fait par une **réutilisation d'un préfixe du prédicat de**

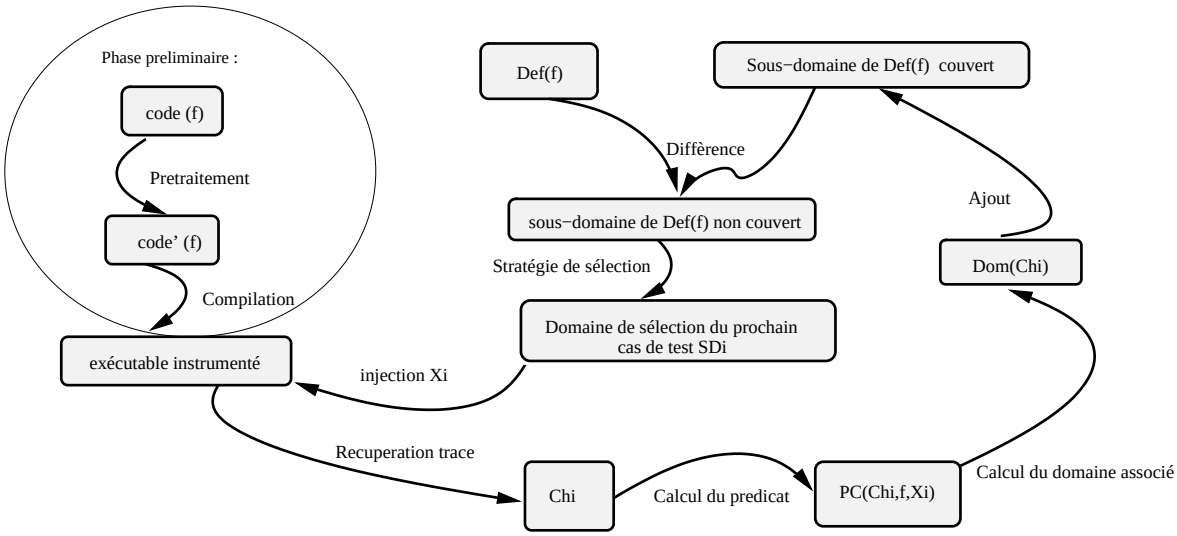

FIG. 7.1 – Principe de base de notre approche

chemin courant afin de limiter le domaine des entrées encore non couvert via une stratégie de diagonalisation telle que

$$Xi \in \neg(Dom(Ch_1) \cup \ldots \cup Dom(Ch_{i-1}))$$

avec X_i le i^{eme} vecteur d'entrée et $Dom(Ch_i)$ le domaine couvert au i^{eme} cas de test.

Il s'agit donc d'une méthode de test **itérative et adaptative**. Cependant, contrairement à la stratégie de Prather et Myers [PM87], nous ne nous limitons pas au critère des branchements mais à un critère plus strict, celui des k-chemins. De plus, notre méthode assure une exploration totale de tous les chemins faisables de la fonction. Ce point sera détaillé dans la section 7.3.1 traitant de la sélection des cas de test par réutilisation de la structure du prédicat de chemin courant.

Notre approche ([Mou03], [Mou04], [WMM03], [WMM04a]) est comparable à une exécution symbolique dynamique. Comme expliqué dans la figure 7.1, la fonction sous test, f, est instrumentée pour récupérer la trace de son exécution sous forme de la trace symbolique du chemin suivi, $\tau(Ch_i)$. Cette trace symbolique permet de calculer le prédicat de chemin correspondant $PC(Ch_i, f, X)$. Le cas de test suivant est ensuite déterminé par réutilisation des informations du prédicat de chemin du cas de test courant dans le domaine des entrées encore non couverts et ainsi de suite jusqu'à ce que tous les chemins (faisables) aient été testés.

7.1.2 Domaines successifs des cas de test

Au début du test, nous disposons du code source de la fonction f sous test ainsi que son domaine de définition $Def(f)$ fournis tous les deux par l'utilisateur.

Notation 14
Le sous-domaine de $Def(f)$ encore non couvert avant de générer le i^{eme} cas de test désigné comme le domaine de sélection du prochain cas de test est noté SD_i avec initialement $SD_0 = Def(f)$.

La première exécution du code instrumenté se fait aléatoirement en choisissant un vecteur d'entrée, X_1 dans SD_0 ce qui permet d'amorcer le test.

L'exécution de X_1 nous permet de définir le prédicat de chemin correspondant $PC(Ch_1, f, X)$. A ce prédicat de chemin correspond le domaine d'entrée $Dom(Ch_1)$ explicitant quels sont les vecteurs X_i dont l'exécution emprunte le chemin Ch_1. Le vecteur d'entrée du prochain cas de test sera choisi dans le domaine de sélection du prochain cas de test SD_1 issue de la différence de SD_0 et du domaine $Dom(Ch_1)$ couvert pour le premier cas de test :

$$SD_1 = SD_0 \setminus Dom(Ch_1) = Def(f) \setminus Dom(Ch_1)$$

Notation 15
Le complémentaire d'un domaine D_1 dans un domaine D_2 sera noté $\overline{D_{1_{(D_2)}}}$ tel que :

$$\overline{D_{1_{(D_2)}}} = D_2 \setminus D_1$$

Ainsi, en reprenant cette notation, $SD_1 = \overline{D_{1_{(Def(f))}}}$.

Convention 5
Pour simplifier les notations, le complémentaire d'un domaine quelconque D dans le domaine de définition de la fonction sous test $Def(f)$ sera noté de façon implicite $\overline{D}$ au lieu de $\overline{D_{(Def(f))}}$.

Nous avons donc, toujours en reprenant cette notation, $SD_1 = \overline{Dom(Ch_1)}$.

Nous pouvons ainsi formaliser le complémentaire de $Dom(Ch_i)$ dans $Def(f)$ noté $\overline{Dom(Ch_i)}$ comme :

$$\overline{Dom(Ch_i)} = \{X \in Def(f) | \neg PC(Ch_i, f, X)\}$$

De façon générale, si n chemins faisables existent, nous pouvons modéliser ainsi les domaines de sélection successifs :

$$\forall i \in 1..n, SD_i = SD_{i-1} \cap \overline{Dom(Ch_{i-1})} = \overline{Dom(Ch_1)} ... \cap \overline{Dom(Ch_{i-1})}$$

ce qui correspond aussi à

$$SD_i = SD_{i-1} \setminus Dom(Ch_i)$$

Quand SD_i est vide cela signifie qu'il n'y a plus de chemins à couvrir (cf. figure 7.2).

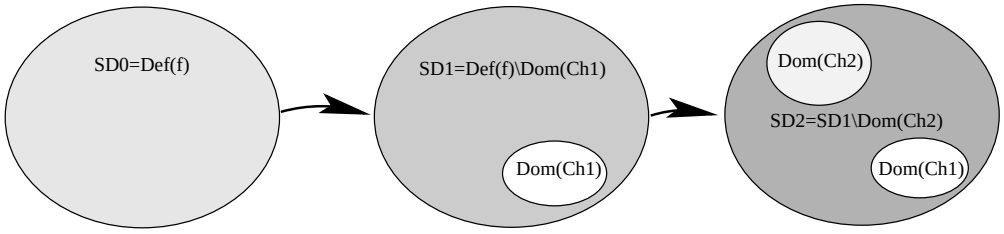

FIG. 7.2 – Domaines des valeurs d'entrée des cas de test

7.1.3 Exploration en profondeur d'abord du CFG

Notre objectif est de garantir l'exploration totale du CFG de la fonction ou du moins de son graphe des chemins faisables. Une **exploration en profondeur d'abord du CFG** permet d'atteindre cet objectif hors présence de chemins de longueur infinie.

Notation 16
Les contraintes successives d'un prédicat de chemin Ch_i seront notées C_i^j où $1 \leq j \leq pc^i$ et notre prédicat est de la forme :

$$(1) PC(Ch_i, f, X) = c_i^1 \wedge ... \wedge c_i^{pc^i}$$

Notation 17
Le plus long préfixe réversible du prédicat de chemin $PC(Ch_i, f, X)$ avec la dernière contrainte niée est noté $MaxC_i(X)$ tel que

$$MaxC_i(X) = (c_i^1 \wedge \ldots \wedge c_i^{j-1} \wedge \neg c_i^j)$$

avec, pour $j \in [2..pc_i], c_i^1 \wedge \ldots \wedge c_i^{j-1} \in PC(Ch_i, f, X)$ et $c_i^j \in PC(Ch_i, f, X)$.

En pratique, la stratégie de sélection des cas de test de PathCrawler réutilise la structure du prédicat de chemin courant pour déterminer $MaxC_i(X)$ caractérisant un sous-domaine de SD_i que nous noterons SSD_i. La détermination du complémentaire des domaines en entrée couverts dans le domaine de définition de la fonction sous test SD_i possède un coût de calcul alors que la stratégie de sélection des cas de test, nous permet quasiment sans calcul de déterminer $MaxC_i$ et donc SSD_i un sous-domaine du complémentaire non couvert. Nous raisonnons donc en pratique sur SSD_i et non sur SD_i comme nous l'avions laissé entendre dans la section précédente.

L'exploration en profondeur d'abord du CFG se fait ainsi en résolvant le système de contraintes $MaxC_i(X)$ correspondant au plus long préfixe du prédicat de chemin courant avec la dernière contrainte niée.

Nous utilisons la PLC (cf. annexe C) pour résoudre $MaxC_i(X)$ correspondant à un CSP (V, D, C) selon la définition C.2.1 avec V l'ensemble des variables d'entrée de la fonction, D les domaines associés à chaque variable d'entrée et C les contraintes de $MaxC_i$ exprimées en fonction des variables d'entrée.

A $MaxC_i(X)$ correspond le domaine SSD_i, sous-domaine en entrée du complémentaire SD_i des domaines d'entrée couverts dans le domaine de définition. Une solution de $MaxC_i(X)$ constituera le cas de test X_{i+1} pour le test suivant avec

$$SSD_i \subseteq SD_i$$

$$SSD_i = \{X \in Def(f) | MaxC_i(X)\}$$

$$X_{i+1} \in SSD_i \rightarrow X_{i+1} \in SD_i$$

Deux remarques importantes sont à faire sur ce point :
- $PC(Ch_{i+1}, f, X)$ le prédicat obtenu par exécution de f pour le vecteur d'entrée X_{i+1} contiendra $MaxC_i(X)$ comme préfixe et
- la négation de $PC(Ch_i, f, X)$ (exprimée selon (1)) contenue dans $MaxC_i(X)$ contient l'exclusion des domaines de chemins précédemment couverts.

Ainsi, nous avons choisi de déterminer le domaine du prochain cas de test en réutilisant la structure du dernier prédicat calculé. L'exploration de l'arbre des chemins faisables d'exécution est contrôlée, elle se fait en profondeur d'abord comme le montre la figure 7.3.

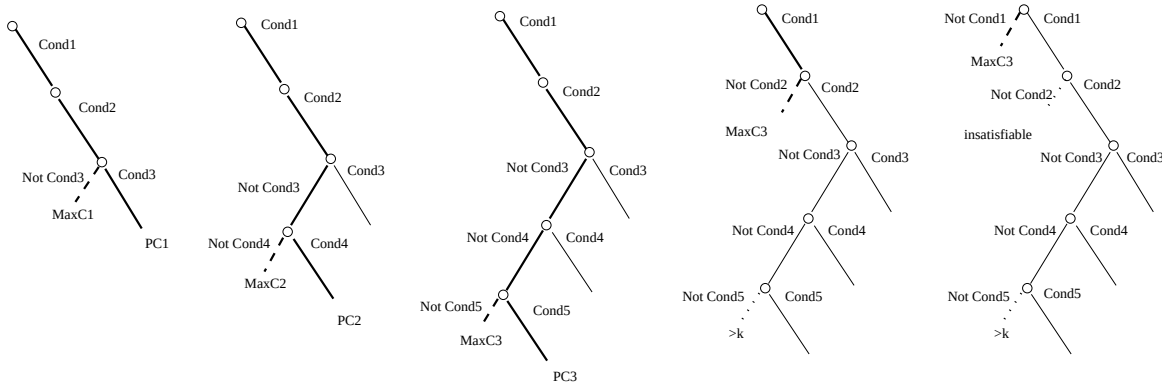

FIG. 7.3 – Illustration de notre stratégie

Notons que l'exploration en profondeur d'abord du graphe prend en compte le critère des k-chemins ce qui signifie que $MaxC_i(X)$ ne contient pas plus de k itérations dans les structures répétitives de la fonction sous test.

Illustration 50
Dans la la figure 7.3, l'exploration de $\neg Cond5$ est arrêtée car elle provoquerait l'exécution d'un chemin à plus de k itérations dans une structure répétitive.

L'avantage de cette stratégie de sélection par réutilisation de la structure du prédicat de chemin courant permet de caractériser facilement un système de contraintes caractérisant un sous-domaine en entrée encore non couvert. Ce sous-domaine en entrée non couvert SSD_i se détermine facilement par notre stratégie. Caractériser à chaque cas de test le complémentaire des domaines des chemins couverts dans le domaine de définition de la fonction aurait été plus complexe à mettre en œuvre. De plus l'information sur les contraintes des prédicats de chemin nous permet de faciliter la détection des chemins infaisables.

Maintenant que nous avons expliqué brièvement le fonctionnement de notre méthode, nous allons aller plus en détail en commençant par le prétraitement du code source de la fonction sous test effectué en amont de la génération des différents cas de test.

7.2 Prétraitement du langage C

Nous étudions des programmes implantés en langage C que nous soumettons dans un premier temps à un analyseur syntaxique CIL [Lan06] qui va transformer le code pour nous donner un programme C équivalent.

7.2.1 Sous-ensemble du langage C ANSI considéré

Le programme de base doit être implanté en ANSI C [KR04] à quelques restrictions près comme :

- les points encore non traités mais dont le traitement est en étude :
  - les fonctions à nombre variable d'arguments,
  - les allocations et libérations dynamiques de mémoire (la difficulté réside dans les possibles structures de données circulaires et le nombre d'alias potentiellement infini en découlant),
  - les appels aux fonctions d'entrée/sortie de la bibliothèque "ANSI Runtime",
  - les types flottants (`float, double,...`), une solution a été proposée dans [BGM06] mais nous ne l'avons pas encore intégrée pour le moment,

  - les conversions de types explicites ou implicites sur des types non entiers et
  - les pointeurs de fonctions.
- les points ne possédant pas de solution connue ou dont le traitement révèle une grande difficulté de mise en place :
  - les sauts de code arrières (instruction goto), limitation imposée par l'utilisation de CIL [Lan06],
  - les fonctions récursives (explosion combinatoire induite et critère des k-chemins non adapté),
  - les structures ou unions de type `void*` (problème de polymorphisme)

Dans la suite, nous excluons l'ensemble des restrictions données ci-dessus.

Nous allons donner ici quelques illustrations des principales modifications du programme de base lors du prétraitement pour obtenir le programme équivalent qui sera soumis à la méthode PathCrawler (cf. figure 7.4).

FIG. 7.4 – Prétraitement du langage C

7.2.2 Décomposition des conditions multiples

Une des premières transformations du code de la fonction sous test effectuée lors du prétraitement concerne la décomposition des conditions multiples du code à savoir les décisions composées de conjonction et/ou disjonction de conditions booléennes que nous désignerons comme conditions élémentaires.

```
1  void condmult(int x1,int x2)
2  {
3    if ((x1>5)||(x2==4)) /*disjonction de conditions élémentaires */
4      x1=12;
5
6    else
7      if((x1>0)&&(x2<0)) /*conjonction de conditions élémentaires */
8        x1=0;
9      else
10       x1=-10;
11 }
```

FIG. 7.5 – Fonction C avec conditions multiples

La fonction équivalente obtenue après prétraitement du code consiste à décomposer ces conditions multiples en une succession de conditions élémentaires.

Illustration 51
La figure 7.5 contient le code d'une fonction C contenant des instructions conditionnelles à conditions multiples. L'équivalent obtenu de la fonction `condmult` *est présenté dans la figure 7.6. Si nous regardons la ligne 3 de la figure 7.5, la condition de la structure conditionnelle est la condition multiple disjonctive «*`((x1>5)||(x2==4))`*» qui sera vérifiée à partir du moment où au moins une*

des conditions élémentaires est vérifiée. Si nous regardons maintenant la transformation de cette condition multiple dans la figure 7.6, nous nous apercevons que celle-ci est décomposée en deux structures conditionnelles imbriquées (cf. les lignes 4 et 10 de la figure). La deuxième structure conditionnelle est contenue dans le bloc correspondant à la non vérification de la première partie de la condition. Cela s'explique par le fait que pour une condition multiple disjonctive, il suffit de vérifier un unique membre de cette condition pour que celle-ci soit vérifiée. Ce point correspond à la sémantique opérationnelle du langage C pour laquelle les sous-conditions sont satisfaites par ordre jusqu'à la satisfaction de la condition multiple complète.

En revanche, pour une condition multiple conjonctive comme à la ligne 7 de la figure 7.5 («`((x1>0)&&(x2<0))`»), il suffit qu'un unique membre ne soit pas vérifié pour que la condition ne le soit pas non plus. Nous nous apercevons ainsi que la transformation de cette condition multiple dans la figure 7.6 correspond à deux structures conditionnelles imbriquées (cf. les lignes 16 et 18 de la figure) mais que, cette fois, la deuxième structure conditionnelle est contenue dans le bloc correspondant à la vérification de la première partie de la condition.

```
1  void condmult(int x1 , int x2 )
2  {
3    {
4      if (x1 > 5) /*première condition de la disjonction*/
5        {
6          x1 = 12;
7        }
8      else
9        {
10         if (x2 == 4) /*seconde condition de la disjonction */
11           {
12             x1 = 12;
13           }
14         else
15           {
16             if (x1 > 0) /*première condition de la conjonction */
17               {
18                 if (x2 < 0) /*seconde condition de la conjonction */
19                   {
20                     x1 = 0;
21                   }
22                 else
23                   {
24                     x1 = -10;
25                   }
26               }
27             else
28               {
29                 x1 = -10;
30               }
31           }
32       }
33     return;
34   }
35 }
```

FIG. 7.6 – Décomposition des conditions multiples

Du fait que toutes les conditions multiples soient décomposées en une succession de conditions élémentaires, la fonction initiale contient moins ou autant de chemins que la fonction obtenue

après prétraitement. L'application du critère des k-chemins sur la fonction transformée assure que le critère est aussi vérifié sur la fonction initiale : couvrir tous les chemins après prétraitement du code revient à couvrir tous les chemins du code initial de la fonction sous test.

Illustration 52
La figure 7.7 contient le CFG de la fonction `condmult` *avant le prétraitement et le CFG de la même fonction après prétraitement. On passe de trois chemins dans le CFG à 5 chemins après décomposition des conditions multiples.*

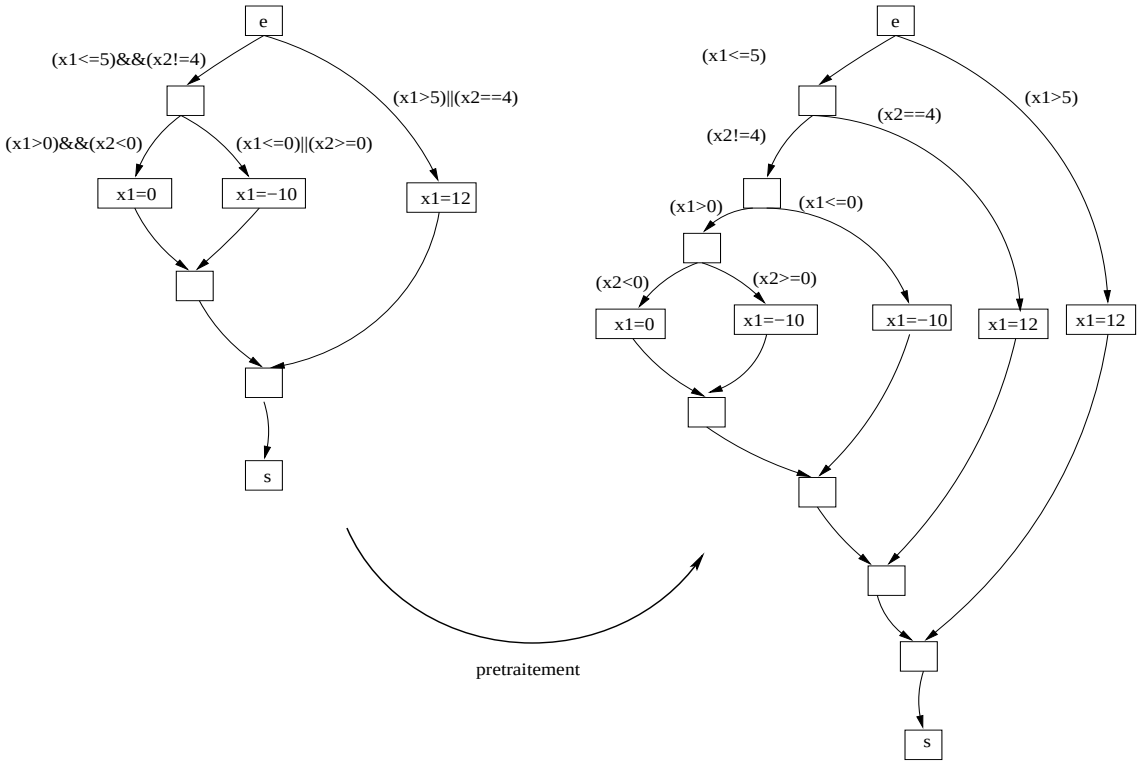

FIG. 7.7 – CFG avant et après prétraitement

Ainsi, notre critère de test, grâce à la décomposition des conditions multiples, permet d'atteindre également le critère MC/DC [CS94].

DÉFINITION – 7.2.1

> *Le critère de couverture* ***MC/DC*** *(«Modified Condition/Decision Coverage») est satisfait si et seulement si toutes les instructions de la fonction ont été couvertes et que pour chaque décision*[1]*, toutes les combinaisons possibles des conditions élémentaires influant sur la valeur de la décision ont également été couvertes au moins une fois.*

Le critère MC/DC est une variante du critère des branchements et l'un des critères les plus utilisés dans les applications industrielles, notamment les applications critiques dans des domaines comme l'aéronautique. Ce critère est imposé, pour certains niveaux de criticité comme pour la norme DO178-B.

[1] Une **décision** est une condition de branchement pouvant soit être composée d'une condition booléenne élémentaire (sans opérateur logique) soit d'une conjonction et/ou disjonction de conditions booléennes élémentaires. Une décision est donc équivalente à une condition élémentaire ou à ce que nous avons appelé une condition multiple.

Illustration 53
Pour la décision suivante : $D = (A \wedge (B \vee C))$, l'application du critère MC/DC impose de trouver pour chaque condition élémentaire, 2 cas de test modifiant la valeur de D pour des valeurs fixées des autres conditions de la décision :

- *pour A :*
  - $A = 0, B = 1, C = 1 \Rightarrow D = 0$
  - $A = 1, B = 1, C = 1 \Rightarrow D = 1$
- *pour B :*
  - $A = 1, B = 1, C = 0 \Rightarrow D = 1$
  - $A = 1, B = 0, C = 0 \Rightarrow D = 0$
- *pour C :*
  - $A = 1, B = 0, C = 1 \Rightarrow D = 1$
  - $A = 1, B = 0, C = 0 \Rightarrow D = 0$ *(correspond au second cas de test pour B)*

Cinq cas de test suffisent pour respecter le critère MC/DC pour la décision D.

Remarque(s) 20
Certains langages, comme le langage C, possèdent des opérateurs "courts-circuits". Si nous prenons la condition composée `(cond1 && cond2)` *évaluée en C, l'opérateur disjonctif n'évalue la seconde condition que si la première est vérifiée. Pour des langages contenant de tels opérateurs, toutes les combinaisons de conditions simples d'une condition multiple ne sont pas évaluées. Le critère MC/DC est alors adapté dans la norme DO178-B comme expliqué dans [CS94].*

Le langage C possédant des opérateurs "courts-circuits", un nombre moindre de cas de test est requis pour atteindre le critère MC/DC (cf. illustration 54). La couverture des chemins du code source prétraité de la fonction sous test permet ainsi de garantir la couverture du critère MC/DC sur le code source initial de la fonction sous test.

Illustration 54
Nous reprenons la décision $D = (A \wedge (B \vee C))$ pour la soumettant à la méthode PathCrawler. Le code source est transformé selon la décomposition des conditions multiples expliquée précédemment. L'application de l'exploration en profondeur d'abord du graphe de contrôle transformé entraîne les cas de test suivants :

- $A = 1, B = 1 \Rightarrow D = 1$
- $A = 1, B = 0, C = 1 \Rightarrow D = 1$
- $A = 1, B = 0, C = 0 \Rightarrow D = 0$
- $A = 0 \Rightarrow D = 0$

Le critère MC/DC adapté aux langages à opérateurs "court-circuit" est atteint.

Nous construisons un jeu de test plus grand que celui du critère des chemins ce qui renforce la rigueur de nos tests. En ce qui concerne une possible perte d'information quant à la couverture du code initial, nous verrons plus tard que lors de l'instrumentation nous récupérons également, pour les conditions de branche, les informations associées dans le code initial de la fonction sous test. Cela nous permet donc de savoir quels chemins de la fonction sous test initiale (avant prétraitement) ont été couverts.

7.2.3 Conflits de variables

Une seconde étape du prétraitement consiste en un renommage des variables afin d'éliminer tout conflit de variables entre les variables locales d'une fonction donnée d'une part et entre les variables locales des fonctions d'un programme et ses variables globales d'autre part.

La fonction équivalente obtenue après prétraitement du code contient les mêmes variables locales à l'exception des celles concernées par le conflit de variables qui ont été renommées. De plus, lors du prétraitement, les déclarations des variables locales sont regroupées et remontées en début de fonction.

```
1  int pbvar(void)
2  {
3    int a ;
4    int i ;
5    char c ;
6    int j ;
7    float i___0 ;/*renommage et remontée de la variable */
8    {
9      a = 1;
10     c = (char )"c";
11     i = 0;
12     {
13       while (1)
14         {
15         while_0_continue:
16           if (i >= 3)
17             {
18               goto while_0_break;
19             }
20           i___0 = (float )0;
21           i___0 = (float )2 * i___0;
22           i ++;
23         }
24     while_0_break:
25     }
26     {
27       return (0);
28     }
29   }
30 }
```

FIG. 7.8 – Élimination des conflits de variables

Illustration 55
Le résultat du prétraitement appliqué à la fonction `pbvar` *de la figure 2.4 page 29 est présenté dans la figure 7.8.*

Les deux variables de même nom, `i`*, déclarées à deux moments différents de la fonction (lignes 5 et 12 de la figure 2.4, page 29) sont renommées (cf. ligne 4 et ligne 7 de la figure 7.8). De plus, toutes les déclarations des variables locales se situent entre les lignes 3 et 7 alors que dans la fonction initiale de la figure 2.4, les déclarations locales étaient étalées dans le corps de la fonction.*

Ainsi, il n'existe plus que des variables locales dont la portée concerne le corps d'une fonction et non des variables locales dont la portée se limite à un bloc interne d'une fonction.

Remarque(s) 21
Comme nous l'avons déjà dit, les conflits de variables sont supprimés entre :
- *les variables locales d'une fonction donnée et*
- *les variables locales et les variables globales d'un même programme.*

Cela signifie donc que deux fonctions d'un même programme peuvent posséder des variables locales de même nom à la condition que ce nom ne soit attribué à aucune des variables globales du même programme.

7.2.4 Forme canonique des structures répétitives

Nous nous apercevons également dans la figure 7.8 précédente que la structure répétitive While (cf. section 2.3.4) initiale a également été transformée. En effet, lors du prétraitement, toutes les structures répétitives du langage sont mises sous une forme canonique.

```
1  void boucle(int x1,int x2)
2  {
3    int i;
4    do /*structure répétitive do while*/
5      {
6        x2=x2/2;
7        x1++;
8      }while (x1<0);
9    while (x2<=15) /*structure répétitive while*/
10     {
11       x1=x1+5;
12       x2++;
13     }
14   for(i=0;i=12;i++) /*structure répétitive for do*/
15     {
16       x1=x1+x2;
17       x2=x1*x2;
18     }
19
20 }
```

FIG. 7.9 – Fonction C avec structures répétitives

Illustration 56
La figure 7.9 contient le code d'une fonction C contenant les différents types de structures répétitives du langage présentées dans la section 2.3.4 et en annexe, à la section A.2. La fonction équivalente obtenue après prétraitement du code transforme toutes ces structures répétitives sous une forme canonique comme le montre la figure 7.10 contenant la fonction équivalente générée.

```
1  void boucle(int x1 , int x2 )
2  {
3    int i ;
4    {
5      {
6        while (1) /*ancienne structure répétitive DoWhile*/
7          {
8            while_0_continue: ;
9            x2 = x2 / 2;
10           x1 ++;
11           if (x1 >= 0)
12             {
13               goto while_0_break;
14             }
15         }
16     while_0_break:;
17     }
18     {
19       while (1) /*ancienne structure répétitive While*/
20         {
21         while_1_continue: ;
22           if (x2 > 15)
23             {
24              goto while_1_break;
25             }
26            x1 =x1 + 5;
27           x2 ++;
28         }
29     while_1_break:;
30     }
31     i = 0;
32     {
33       while (1) /*ancienne structure répétitive ForDo*/
34         {
35         while_2_continue: ;
36           if (i > 12)
37             {
38             goto while_2_break;
39             }
40          x1 =x1 + x2;
41           x2 =x1 * x2;
42           i ++;
43         }
44     while_2_break:;
45     }
46     return;
47   }
48 }
```

FIG. 7.10 – Mise sous forme canonique des structures répétitives

Nous ne nous attarderons pas ici à détailler la transformation de chaque type de structure répétitive. Nous faisons simplement remarquer que la forme canonique associée par le prétraitement consiste en la création d'une structure répétitive infinie (`while(1)`) contenant une structure conditionnelle dont la condition est celle de la structure répétitive initiale et dont la vérification entraîne l'exécution du corps de la structure répétitive et la non vérification entraîne au saut dans

le code à l'extérieur de la structure répétitive via l'utilisation de l'instruction `goto`. Pour de plus amples détails, le lecteur pourra se reporter à [Lan06].

7.2.5 Transformation des structures conditionnelles Switch et SwitchBreak

Les structures conditionnelles ne sont, quant à elles, pas modifiées par le prétraitement à l'exception de celles contenant des conditions multiples et des structures conditionnelles Switch et SwitchBreak présentées en annexe dans la section A.1.

```
1  int main() {
2    int result;
3    switch (i) {  /*début de la structure conditionnelle SwitchBreak */
4    case 0 :
5      result=0;
6      break;
7    case 1 :
8      result=1;
9      break;
10   case 2 :
11     result=2;
12     break;
13   default :
14     result=0;
15   }
16   return(result);
17 }
```

FIG. 7.11 – Exemple d'utilisation d'une structure conditionnelle SwitchBreak

Illustration 57
La figure 7.11 contient le code d'une fonction C avec une structure conditionnelle SwitchBreak qui a été décomposée en une imbrication de structures conditionnelles IfThenElse (cf. figure 7.12).

```
1  int main(void)
2  {
3    int i ;
4    int result ;
5    if (i == 0) /*ancienne structure conditionnelle Switch*/
6      {
7        goto switch_0_0;
8      }
9    else
10     {
11       if (i == 1)
12         {
13           goto switch_0_1;
14         }
15       else
16         {
17           if (i == 2)
18             {
19               goto switch_0_2;
20             }
21           else
22             {
23               {
24                 goto switch_0_default;
25                 if (0)
26                   {
27                   switch_0_0:
28                     result = 0;
29                     goto switch_0_break;
30                   switch_0_1:
31                     result = 1;
32                     goto switch_0_break;
33                   switch_0_2:
34                     result = 2;
35                     goto switch_0_break;
36                   switch_0_default:
37                     result = 0;
38                   }
39                 else
40                   {
41                   switch_0_break:;
42                   }
43               }
44             }
45         }
46     }
47   return (result);
48 }
```

FIG. 7.12 – Transformation d'une structure conditionnelle SwitchBreak

7.2.6 Isolement des expressions à effets de bord

Une autre transformation non négligeable faite pendant le prétraitement concerne les expressions à effets de bord. Après prétraitement, chaque instruction contient au plus une expression à effets de bord via l'insertion de nouvelles variables.

```
1  int f(int x)
2  {
3    return (x ++ + g(x));
4  }
```

FIG. 7.13 – Fonction avec expressions à effets de bords

```
1  int f(int x)
2  {
3    int tmp ;
4    int tmp___0 ;
5    {
6    tmp = x;
7    x ++;
8    tmp___0 = g(x);
9    return (tmp + tmp___0);
10   }
11 }
```

FIG. 7.14 – Isolement des expressions à effets de bord

Illustration 58
Si nous regardons le code source de la figure 7.13 à la ligne 3, nous pouvons observer une instruction consistant en la somme de deux expressions à effets de bord : `x++` et `g(x)`. Cette instruction est décomposée en deux instructions contenant chacune une des expressions à effets de bord (cf. figure 7.14 aux lignes 7 à 9).

Cette transformation permet une meilleure analyse de la fonction sous test : les effets de bords sont plus facilement identifiables ce qui nous permet de connaître plus facilement la valeur d'une variable à chaque instant.

7.2.7 Instrumentation

Chaque bloc élémentaire d'affectations et condition du code source sont transformées sous forme de termes Prolog (cf. figure 7.1) contenant des annotations supplémentaires pour les conditions de structures répétitives utilisées pour la mise en œuvre du critère des k-chemins. Lors du prétraitement de la fonction, une table d'association (contenue dans un programme Prolog généré automatiquement) est créée automatiquement pour donner pour chaque branchement la condition et le bloc élémentaire d'affectations séquentiel correspondant. Nous ajoutons en parallèle, dans le code source transformé, les instructions de trace contenant les identifiants des conditions et blocs élémentaires d'affectations vérifiés lors de l'exécution de la fonction dans le but modéliser le chemin suivi en un système de contraintes logiques du premier ordre.

Table d'association

Un programme Prolog (notre table d'association contenue dans le programme Prolog généré automatiquement) est ainsi généré lors de l'instrumentation contenant les différents termes correspondant aux affectations $block\_assignments(id_a, [termes_a])$ et aux différentes conditions du code $block\_condition(id_c, clause_c, a(NL, NumCond, fonc, neg/pos, [annot\_boucle]))$ avec :

- id_a l'identifiant des affectations C associées et $[clause_a]$ la liste des termes équivalents Prolog,
- id_c l'identifiant de la condition C associée et $clause_c$ son équivalence en termes Prolog,
- $a(NL, NumCond, fonc, neg/pos, [annot\_boucle])$ l'annotation associée aux conditions avec :
  - NL le numéro de la ligne du code source initiale contenant cette condition,

- $NumCond$ le numéro de cette condition à la ligne NL pour le cas de conditions multiples,
- $fonc$ le nom de la fonction source,
- pos si la dite condition a été vérifiée ou neg sinon et
- [$annot\_boucle$] l'annotation associée aux conditions de structure répétitive que nous verrons plus en détails dans le chapitre 8.

Remarque(s) 22
Dans les instructions de trace, nous n'imprimons pas directement les termes Prolog associés mais leurs identifiants. La correspondance se fait donc via le programme Prolog généré.

Les problèmes possibles quant à une instrumentation sont les suivants :
- la taille du code est plus conséquente (facteur 4 en général),
- l'instrumentation ne doit pas modifier le comportement du composant sous test.

Représentation canonique des noms de variables et chemins d'accès

La complexité du langage C nécessite une uniformisation des noms de variables et chemins d'accès par rapport aux différentes notations possibles, en particulier pour les tableaux et pointeurs mais aussi pour les structures, les unions,.... Nous avons donc fixé certaines formes canoniques pour l'écriture de ces variables et chemins d'accès. La figure 7.1 donne le formatage des différents types de référence aux variables du langage C utilisées dans les termes Prolog associés.

Type de variable ou de chemin	Code C correspondant	Formalisation Prolog
variable simple	`var`	var
tableau	`tab[n]`	$cont(tab, n)$
matrice	`mat[n][p]`	$cont(cont(tab, n), p)$
structure	`struct.champ`	$cont(struct, i).is\_struc(struct, champ, i)$
structure pointée	`struct->champ`	$cont(cont(struct, 0), i).is\_struc(cont(struct, 0), champ, i)$
union	`union.champ`	$cont(union, i).is\_struc(union, champ, i)$
union pointée	`union->champ`	$cont(cont(union, 0), i).is\_struc(cont(union, 0), champ, i)$
adresse de la variable	`&var`	$ad(var)$
contenu d'un pointeur	`*var`	$cont(var, 0)$
adresse du premier élément d'un tableau	`&tab[0]`	$ad(cont(tab, 0))$
adresse du premier élément d'une matrice	`&mat[0][0]`	$ad(cont(cont(mat, 0), 0))$

TAB. 7.1 – Formatage des variables dans les termes Prolog

Grâce à la table d'association générée, la trace du chemin suivi lors de l'exécution est transformée en termes Prolog dont la figure 7.2 en donne la signification.

Illustration 59
$[\![$`X=2*Y+1`$]\!] \equiv [\![affect(X, +(mult(2, Y), 1))]\!]$
$[\![$`((X-2) > 5)`$]\!] \equiv [\![cond(sup, -(X, 2), 5)]\!]$

Notre méthode traite encore bien d'autres opérations (opérateurs logiques ou binaires,...) mais nous avons préféré ne présenter dans ce tableau que les termes Prolog correspondant aux cas les plus courants.

Code C correspondant	Termes Prolog	Correspondance langage naturel
`E1=E2`	$affect(E1, E2)$	Affectation de la valeur de l'expression E2 à l'expression E1
`if (E1 Rel E2)`	$cond(Rel, E1, E2)$	Condition : la relation Rel est vraie entre E1 et E2, Rel appartenant à $>$, $>=$, $<$, $<=$, $!=$, $==$
`>`	sup	supériorité stricte
`>=`	$supegal$	supériorité non stricte
`!=`	$diff$	inégalité stricte
`==`	$egal$	égalité
`<`	inf	infériorité stricte
`<=`	$infegal$	infériorité non stricte
`E1*E2`	$mult(E1, E2)$	multiplication
`E1/E2`	$div(E1, E2)$	division entière
`E1+E2`	$+(E1, E2)$	addition
`E1-E2`	$-(E1, E2)$	soustraction
`E1%E2`	$mod(E1, E2)$	reste de la division entière

TAB. 7.2 – Équivalence de la syntaxe C et de la syntaxe Prolog

Ces termes Prolog sont uniquement exprimés en fonction de noms canoniques de variables et de constantes. De plus, les expressions comportant des opérations avec des constantes sont simplifiées par un processus d'évaluation partielle lors de l'instrumentation.

Simplification des termes Prolog

Des règles de simplification sont aussi appliquées lors de la création de la table d'association (cf. tableau 7.3). Le but est de réduire la difficulté d'analyse lors de la substitution implantée en ECLiPSe.

D'autres simplifications sont également possibles pour la réécriture de chemins complexes d'accès à une variable comme par exemple les règles similaires à la deuxième et troisième ligne du tableau 7.3 appliquée à la soustraction, la multiplication ou la division entière. Une telle simplification facilitera l'analyse ultérieure des alias .

Notation 18
La notation $exp1 \rightsquigarrow exp2$ *signifie que l'application des règles de simplification sur l'expression* $exp1$ *a pour résultat l'expression* $exp2$.

Code C	Réécriture en termes Prolog	Simplification des termes	Règles appliquées
`*&var`	$cont(ad(var), 0)$	var	$cont(ad(x), 0) \rightsquigarrow x$
`&tab+2`	$+(ad(cont(tab, 0)), 2)$	$ad(cont(tab, +(0, 2))) \equiv ad(cont(tab, 2))$	$+(ad(x), c) \rightsquigarrow ad(+(x, c))$
`3+12`	$+(3, 12)$	15	$+(n1, n2) \rightsquigarrow n3$ où $n3$ est le résultat de l'évaluation de la somme de $n1$ et $n2$

TAB. 7.3 – Exemples de règles de simplification des termes Prolog

Les règles de simplification données sur les termes Prolog faisant intervenir l'addition de la figure 7.3 s'appliquent de façon identique à tous les opérateurs présentés dans la figure 7.2.

Illustration 60
La clause $+(2, +(B, mult(3 * 5)))$ *est simplifiée en* $+(17, B)$.

Remarque(s) 23
Les règles de simplification sont appliquées aussi lors de la substitution.

Annotation des contraintes

La combinatoire des chemins due aux structures conditionnelles et en particulier répétitives nous oblige souvent à restreindre notre critère à celui des k-chemins.

Pour respecter ce critère de test, nous avons mis en place, lors de l'instrumentation, une annotation des conditions associée aux structures répétitives telle que chaque contrainte est associée à sa nature dans le code à savoir :
- une entrée de structure répétitive,
- une sortie de structure répétitive,
- la première condition d'entrée de structure répétitive pour une condition multiple,
- ...

Ces informations seront réutilisées par la suite lors de notre stratégie de sélection des cas de test expliquée dans la section 7.3.2.

7.3 Notre approche

La méthode se déroule donc en quatre étapes principales :

1. le prétraitement, la création du lanceur de test et la compilation de la fonction sous test sont exécutés en début de test,
2. un choix aléatoire d'une donnée de test X_i dans le domaine de sélection SSD_{i-1} (sous-domaine de $Def(f)$ encore non couvert),
3. le calcul du prédicat $PC(Ch_i, f, X)$ via la trace symbolique d'exécution $\tau(Ch_i)$ récupérée définissant le domaine en entrée couvert pour ce cas de test $Dom(Ch_i)$ et de SSD_i le complémentaire de $Dom(Ch_i)$ dans SSD_{i-1}
4. la résolution des contraintes $MaxC_i(X)$ (selon notre stratégie de choix de cas de test pour déterminer la prochaine donnée de test X_{i+1} dans le domaine de sélection défini à l'étape précédente SSD_i et répétition des étapes 2 à 4 jusqu'à une couverture de 100% des chemins faisables de la fonction ou la caractérisation d'un domaine de sélection vide.

Contrairement à la majorité des autres méthodes dynamiques de génération de cas de test, nous n'utilisons pas un algorithme de recherche heuristique pour la résolution des contraintes mais la PLC (cf. annexe C), habituellement utilisée pour des méthodes statiques. De ce fait, nous qualifions notre méthode comme une méthode hybride dans la mesure où elle utilise des techniques de méthode dynamique et de méthode statique de génération de cas de test ([WMMR05], [WMM04a]).

7.3.1 Calcul du prédicat de chemin

Nous devons donc définir le prédicat de chemin correspondant à la trace symbolique d'exécution récupérée en substituant les variables de sortie par leur expression en termes des valeurs initiales des variables d'entrée.

Dans le meilleur des cas (c'est-à-dire sans alias), une simple méthode de substitution par remontée dans le code est suffisante (cf. figure 4.6 du chapitre 4).

Cependant, cette méthode simple de calcul de prédicat de chemin est insuffisante dans le cas général.

Pour pallier ce problème, nous avons mis au point une méthode de substitution disposant d'une modélisation de la mémoire permettant d'identifier toutes les correspondances d'alias à chaque instant de l'exécution du code.

Le traitement des alias est simplifié par rapport à d'autres méthodes car nous ne traitons qu'un seul chemin d'exécution à la fois qui est, de plus, totalement déroulé. Nous n'avons donc pas besoin de passer sous une forme intermédiaire du type SSA pour lever l'ambiguïté d'une valeur ni par un traitement des pointeurs du type points-to (cf. section 6.2.2). De plus, dans notre modèle mémoire, toutes les instructions de la fonction sont ordonnées selon leur ordre d'exécution dans le code ce qui nous permet d'annoter la valeur symbolique de chaque variable par un numéro indiquant l'ordre des annotations.

Convention 6
Notre calcul des prédicats de chemin repose sur l'hypothèse suivante :
deux variables d'entrée de type pointeur ne peuvent pas pointer sur des structures de données ayant des adresses mémoire en commun. Cette hypothèse écarte uniquement un type d'alias précis.

Lors de la lecture de la valeur symbolique d'une variable dont le chemin d'accès est indexé par des paramètres d'entrée de la fonction sous test, les éventuelles contraintes sur ces paramètres d'entrée sont également récupérées du modèle de la mémoire. Ainsi, en cas d'utilisation des variables d'entrée, dans un chemin d'accès, tous les cas possibles de relations sur les variables d'entrée influant sur le chemin suivi sont considérés.

Illustration 61
Prenons comme cas de figure la trace symbolique récupérée suivante :
$\tau(Ch_i) = (v = x * 2, v = v - y, pt = \&tab[2], tab[2] = x, *(pt + v) = y, tab[y + 4] > 5, tab[1] = 10)$
avec pour variables d'entrée : $p_1(X) = x$, $p_2(X) = y$ *et* $p_3(X) = z$ *après substitution et mise sous forme canonique, nous obtenons :*
$(v = x*2, v = x*2-y, pt = tab+2, tab[2] = x, *(tab+2+x*2-y) = y, tab[y+4] > 5, tab[1] = 10)$
La condition $tab[y + 4] > 5$ *est vérifiée mais trois scénarii peuvent être distingués dans le prédicat de chemin calculé :*

1. $*(tab + 2 + x * 2 - y) = y$ *est équivalent à* $*(tab + y + 4) = y$,
2. $tab[2] = x$ *est équivalent à* $tab[y + 4] = x$ *et*
3. *les deux scénarii précédents ne sont pas vérifiés et* $tab[y + 4] > 5$.

Les contraintes associées à ces scénarii sont, en respectant le même ordre :

1. $((y + 4 = 2 + x * 2 - y) \wedge (y > 5)) \vee$
2. $((y + 4 \neq 2 + x * 2 - y) \wedge (y + 4 = 2) \wedge (x > 5)) \vee$
3. $((y + 4 \neq 2 + x * 2 - y) \wedge (y + 4 \neq 2) \wedge (tab[y + 4] > 5))$

Dans un tel cas, nous utilisons les valeurs réelles des variables d'entrée pour identifier précisément quel cas est celui correspondant à ce cas de test et les autres scénarii seront explorés automatiquement lors de la négation de ces contraintes d'alias du cas de test donnant lieu à ce chemin ce qui nous permet d'éliminer la disjonction dans nos prédicats de chemin.

7.3.2 Propriétés et utilisation des prédicats de chemin dans PathCrawler

Cette partie est primordiale pour la suite car toute notre méthode de test est basée sur le calcul et la réutilisation des prédicats de chemins.

Nous rappelons que chaque prédicat, $PC(Ch_i, f, X)$, est une conjonction de pc^i conditions de branches (projetées sur les entrées) ou contraintes d'alias successives rencontrées le long du chemin correspondant. Ces conditions de branche sont ordonnées par l'ordre d'exécution des conditions correspondantes. Nous conservons cette conjonction ordonnée dépliée dans la mesure où cela fournit une information importante sur la faisabilité des chemins que nous allons exploiter pour notre stratégie de sélection des cas de test.

Le rôle des prédicats de chemins étant essentiel pour notre méthode, il est donc nécessaire de retenir les propriétés suivantes pour le prédicat de chemin du cas de test courant :

- le prédicat de chemin courant est calculé en partant de la trace symbolique d'exécution du cas de test courant qui correspond à un chemin d'exécution faisable et par conséquent le **système de contraintes associé au prédicat de chemin est toujours satisfiable**,
- **tous les préfixes de ce prédicat sont satisfiables** (puisque la totalité de la conjonction du prédicat est faisable).

De plus, le prétraitement des conditions multiples dans l'implantation de la fonction sous test entraîne que le CFG modifié de la fonction sous test ne possède que des conditions de branches simples. Le prédicat de chemin correspondant à la conjonction des prédicats de branche cela signifie que les prédicats de chemins manipulés dans la méthode PathCrawler sont des conjonctions de conditions simples (les prédicats de chemin ne possèdent aucune disjonction).

La négation de $PC(Ch_i, f, X)$ exprimé sous la forme (1) à la page 101 correspond à la disjonction de tous ses préfixes avec la dernière condition niée[2] :

$$(2) : \neg PC(Ch_i, f, X) = \bigvee_{j=2..pc_i} (c_i^1 \wedge \ldots \wedge c_i^{j-1} \wedge \neg c_i^j) \vee \neg c_i^1$$

Illustration 62
Pour illustrer la formule (2), supposons que le prédicat $PC(Ch_1, f, X)$ du premier cas de test soit composé de deux contraintes ordonnées tel que $PC(Ch_1, f, X) = c_1^1 \wedge c_1^2$. Le domaine de sélection du prochain cas de test correspond donc à $SSD_1 \subseteq \overline{Dom(Ch_1)}$. La négation du prédicat $PC(Ch_1, f, X)$, ce qui donne :

$$\neg PC(Ch_1, f, X) = \neg(c_1^1 \wedge c_1^2) = (\neg c_1^1 \wedge c_1^2) \vee (c_1^1 \wedge \neg c_1^2) \vee (\neg c_1^1 \wedge \neg c_1^2) = \neg c_1^1 \vee (c_1^1 \wedge \neg c_1^2)$$

ce qui correspond bien à l'expression de (2).

Remarque(s) 24
Notons que chaque conjonction de la disjonction (2) correspond à un préfixe de chemin du CFG encore non exploré à la i^{eme} étape de notre stratégie de test.

Application du critère des k-chemins

La définition de $MaxC_i(X)$ est modifiée pour prendre en compte les annotations des contraintes mises en place lors de l'instrumentation plus précisément la contrainte imposée par le critère de test à savoir un nombre maximal d'itérations dans une structure répétitive limité à k.

[2] Nous rappelons que l'expression des prédicats de chemins sous la forme (1) contient une information sur l'ordre de vérification des contraintes contenues dans le prédicats : la contrainte c_i^1 est vérifiée avant la contrainte c_i^2 et la vérification de c_i^1 amène à évaluer la contrainte c_i^2.

Ainsi, si une condition correspond à la tête d'une structure répétitive après k itérations et si elle est la condition d'entrée de structure répétitive (par exemple la dernière condition à vérifier pour une condition composée conjonctive) alors sa vérification entraîne une nouvelle entrée dans la structure répétitive après plus de k itérations. Dans un tel cas, nous coupons donc la branche associée de l'arbre de recherche.

Dans le cas d'une condition multiple d'entrée de structure répétitive, nous déterminons donc le rôle, déterminant ou non pour l'entrée ou la sortie de structure répétitive grâce aux informations tirées de l'instrumentation et transmises aux contraintes concernées.

Cependant, il n'est pas exclu qu'une solution au système de contraintes du domaine de sélection d'un prochain cas de test puisse provoquer une nouvelle entrée dans la structure répétitive après k itérations.

En effet, le vecteur d'entrée du prochain cas de test sera une solution au système de contraintes associé à SSD_{i+1}. Nous garantissons que $MaxC_i(X)$ ne contient pas plus de k itérations par structure répétitive mais nous ne pouvons pas garantir que le prédicat d'un autre cas de test, solution de $MaxC_i(X)$, n'entraînera pas plus de k itérations de cette même structure répétitive ou d'une autre structure répétitive.

Problème des chemins infaisables

En ce qui concerne le problème des chemins infaisables, notre méthode facilite leur détection. Cela s'explique par trois caractéristiques de notre méthode de test :

- nous ne traitons qu'un seul chemin d'exécution de la fonction sous test à la fois,
- de plus, ce chemin est totalement déroulé[3] et enfin
- chaque nouveau cas de test est déterminé comme la solution d'un système de contrainte composé d'un préfixe connu comme satisfiable avec la négation de sa dernière contrainte.

Ainsi, si le système de contraintes de $MaxC_i(X)$ est insatisfiable, nous savons que cette insatisfiabilité provient de l'ajout au préfixe satisfiable de $\neg c_i^j$ (cf. formule (2)).

Nous pouvons donc utiliser, dans la phase de labelling (cf. annexe C) cette information :

- dans le meilleur des cas, il s'agit de confronter la contrainte niée à chaque contrainte du préfixe satisfiable afin de mettre en évidence l'insatisfiabilité du chemin,
- sinon nous utilisons le calcul des dépendances[4] des variables de la dernière contrainte. Dans ce cas, l'ensemble des contraintes faisant intervenir une variable dépendante d'une des variables de la dernière contrainte est confrontée à la dernière contrainte niée et
- dans le pire des cas (si toutes les variables sont concernées par la négation[5]) la complexité devient exponentielle soit par rapport au nombre de contraintes, soit par rapport au nombre de variables dans les contraintes soit par rapport à la taille des domaines énumérés et nous ne gagnons rien.

Nous avons donc mis en place la **gestion d'un TimeOut pour les insatisfiabilités non détectées**, la recherche est arrêtée après le dépassement du TimeOut et est reportée à la fin de la génération des cas de test. Cependant, les expériences faites sur divers exemples confirment la rareté des TimeOut. Cela nous conforte dans l'idée d'une **bonne détection des chemins infaisables**.

[3] La trace du chemin exécuté récupéré correspond à un unique flot de contrôle et contient donc la séquence de toutes les affectations et conditions vérifiées pour ce chemin (indépendamment du fait qu'elles appartiennent à une structure répétitive, conditionnelle ou à un bloc d'instructions élémentaire).

[4] Deux variables sont dites dépendantes si une des deux variables est définie par l'utilisation de la valeur de la seconde variable ou si une des deux variables en définie par l'utilisation d'une troisième variable dont la valeur est dépendante de la valeur de la seconde variable.

[5] Toutes les variables du système de contraintes sont dépendantes des variables de la contrainte niée.

La satisfiabilité de contraintes formées de conjonction de contraintes arithmétiques booléennes sur des domaines finis d'entiers est décidable mais NP-complet. Elle est implantée en programmation logique avec contraintes dans l'environnement ECLiPSe [WNJ97] et en partant de la procédure de résolution utilisée dans [MA00] et dans [GDGM01]. Nous utilisons les heuristiques de labelling suivantes :

- la stratégie appliquée pour les valeurs des variables représentant des dimensions d'autres variables consiste à choisir les valeurs les plus basses possibles,
- nous cherchons dans un premier temps les valeurs des variables contraintes (autre que les variables représentant des dimensions de structures de données) pour lesquelles nous choisissons une valeur aléatoire orientée vers le milieu des domaines courant après l'application de la propagation des contraintes et
- nous choisissons aussi en premier d'instancier les variables les plus contraintes du système.

L'avantage du premier point est de tester des structures à dimension nulle dont le traitement est souvent source d'erreurs. De plus, le temps d'exécution est réduit dans la mesure où un nombre faible d'éléments des variables structurées est généré et utilisé.

Le fait que les dimensions de variables structurées soient choisies de valeur faible permet souvent de limiter le nombre d'itérations des structures répétitives comme celles correspondant à un parcours de ces variables structurées. Nous limitons les cas de test superflus correspondant à un chemin contenant plus de k itérations dans une structure répétitive donnée.

Notre stratégie de choix de cas de test est efficace car elle tire profit du mécanisme du backtrack de la PLC. De plus, l'espace de recherche peut encore être réduit en prenant en compte d'autres contraintes lors de la résolution comme des contraintes fournies par l'utilisateur pour satisfaire un objectif de test donné.

7.4 Stratégie de résolution de PathCrawler

La stratégie de résolution présentée ici est inspirée du solveur de contraintes de l'outil GATeL [MA00] implantée dans l'environnement ECLiPSe [WNJ97].

Notre heuristique de "labelling" (cf. annexe C) consiste à choisir des dimensions de variables structurées les plus petites possibles. Ainsi, nous sommes surs de générer des cas de test pour des structures de données de taille nulle qui sont souvent source d'erreurs (quand elles sont autorisées). De plus, il arrive souvent que le nombre d'itérations d'une structure répétitive soit lié à la dimension d'une variable structurée. Par conséquent, pour de telles structures répétitives, nous limitons de façon indirecte le nombre d'itérations. Pour les variables ne représentant pas des dimensions, un générateur de données aléatoire est utilisé dans le tiers médian des domaines des variables après propagation de contraintes. Si toutes les valeurs du tiers médian n'aboutissent à aucune solution alors les autres valeurs du domaine sont également explorées.

Remarque(s) 25
Les CSP manipulés correspondent aux prédicats de chemins courants dont la dernière contrainte à été niée soit $MaxC_i(X)$. Une solution d'un tel CSP correspond à une donnée d'entrée pour le prochain cas de test.

Le choix de la variable à instancier correspond à une variable associée à un maximum de contraintes et dont le domaine de valeurs est le plus petit. Nous choisissons également à ce moment d'instancier en priorité les variables concernées par la dernière contrainte niée du prédicat de chemin courant.

Ce processus est itéré jusqu'à la satisfaction de toutes les contraintes du CSP en vue d'une affectation totale consistante.

Cette stratégie de randomisation permet d'avoir une solution différente à chaque résolution d'un prédicat de chemin représenté par un CSP. Pour la sélection d'un cas de test, lors de la

résolution du préfixe de prédicat de chemin courant avec la dernière condition niée (modélisé sous la forme d'un CSP), les données d'entrée déterminées ne seront pas toujours les mêmes valeurs. Ce point nous permet de vérifier l'efficacité et la constance de notre méthode de génération de cas de test unitaire pour des données de test générées variables.

Nous allons maintenant illustrer concrètement notre méthode sur un premier exemple.

7.5 Application à la fonction `getmid`

Nous avons choisi de reprendre la fonction `getmid` de la figure 6.2, page 85 prenant en arguments trois entiers et retournant la valeur intermédiaire de ceux-ci.

Nous allons suivre l'application de notre méthode sur cette fonction au fil des différents cas de test.

7.5.1 Remarques préalables

Tout d'abord, observons le CFG de cette fonction, représenté par la figure 6.3 à la page 85.

Afin de faciliter la compréhension de l'exemple, nous rappelons que chaque arc est annoté d'un numéro unique de type ci avec $c_i = \neg c_{i+1}$ pour i impair $\in [1..11]$ étiqueté de la condition de branche associée. Le graphe contient au total 16 chemins dont certains sont en réalité infaisables. En effet, pour σ_N la fonction d'étiquetage associée au nœuds du graphe de contrôle de la fonction, nous avons :

- $\sigma_N(c_2) = \neg\sigma_N(c_8)$ et
- $\sigma_N(c_1) = \neg\sigma_N(c_7)$.

Les contraintes associées aux nœuds c_2 et c_8 sont contradictoires. Tout chemin passant à la fois par c_2 et c_8 ou c_1 et c_7 sera donc infaisable puisque son prédicat est insatisfiable.

Remarque(s) 26
Pour cet exemple, nous allons confondre les arcs du CFG avec les labels associés ainsi c_j désignera à la fois l'arc c_j et sa contrainte associée $\sigma_N(c_j)$.

La figure 7.15 contient la fonction équivalente instrumentée lors du prétraitement et la figure 7.16 contient le programme généré contenant les termes Prolog correspondant aux affectations et conditions vérifiées.

Lors de l'exécution de la fonction instrumentée, les traces récupérées sont de la forme : "$.id1 : id2 : id3.id4$" où les identifiants précédés d'un point comme $id1$ et $id4$ sont des identifiants correspondant aux termes *block_condition* et les identifiants précédés de deux points sont des identifiants correspondant aux termes *block_assignments* du programme Prolog généré.

7.5.2 Application

Nous devons tout d'abord déterminer le domaine de définition $Def(getmid)$ de la fonction. Aucune contrainte n'est imposée entre les différentes variables d'entrée donc :

$$Def(getmid) = Dom(int) \times Dom(int) \times Dom(int)$$

Nous choisissons pour cet exemple de restreindre les domaines associés aux variables d'entrée de la fonction, possibilité aussi offerte à l'utilisateur pour orienter le test de la fonction sur un sous-domaine en entrée de la fonction jugé critique.

Notation 19
Comme nous venons de le préciser, l'utilisateur peut restreindre les domaines des variables d'entrée pour orienter les tests. Le domaine associé à la fonction $Dom(f)$ peut donc être restreint par l'utilisateur à un sous-domaine que nous noterons $Dom(f)_{|user}$ tel que $Dom(f)_{|user} \subseteq Dom(f)$. Dans ce cas, le domaine de sélection du premier cas de test est également restreint à un sous-domaine du domaine de définition de la fonction que nous noterons $Def(f)_{|user}$ tel que $Def(f)_{|user} \subseteq Def(f)$.

```
1  int getmid(int x1 , int x2 , int x3 )
2  { int mid ;
3    {
4    printf(":1");
5    mid = x3;
6    if (x2 < x3) {
7      printf(".2");
8      if (x1 < x2) {
9        printf(".4:6");
10       mid = x2;
11     } else {
12       printf(".5");
13       if (x1 < x3) {
14         printf(".7:9");
15         mid = x1;
16       } else {
17         printf(".8");
18       }
19     }
20   } else {
21     printf(".3");
22   }
23   if (x2 >= x3) {
24     printf(".10");
25     if (x1 > x2) {
26       printf(".12:14");
27       mid = x2;
28     } else {
29       printf(".13");
30       if (x1 > x3) {
31         printf(".15:17");
32         mid = x1;
33       } else {
34         printf(".16");
35       }
36     }
37   } else {
38     printf(".11");
39   }
40   {
41   printf(":18");
42   return (mid);
43   }
44 }}
```

FIG. 7.15 – Fonction `getmid` après prétraitement dont instrumentation

```
1  block_assignments(18,[affect(res__getmid,mid)]).
2  block_assignments(17,[affect(mid, x1)]).
3  block_assignments(16,[]).
4  block_assignments(15,[]).
5  block_assignments(14,[affect(mid, x2)]).
6  block_assignments(13,[]).
7  block_assignments(12,[]).
8  block_assignments(11,[]).
9  block_assignments(10,[]).
10 block_assignments(9,[affect(mid, x1)]).
11 block_assignments(8,[]).
12 block_assignments(7,[]).
13 block_assignments(6,[affect(mid, x2)]).
14 block_assignments(5,[]).
15 block_assignments(4,[]).
16 block_assignments(3,[]).
17 block_assignments(2,[]).
18 block_assignments(1,[affect(mid, x3)]).
19 block_assignments(0,[]).
20
21 block_condition(16,cond(infegal,x1,x3,a(18,1,getmid.c,neg,[]))).
22 block_condition(15,cond(sup,x1,x3,a(18,1,getmid.c,pos,[]))).
23 block_condition(13,cond(infegal,x1,x2,a(15,1,getmid.c,neg,[]))).
24 block_condition(12,cond(sup,x1,x2,a(15,1,getmid.c,pos,[]))).
25 block_condition(11,cond(inf,x2,x3,a(13,1,getmid.c,neg,[]))).
26 block_condition(10,cond(supegal,x2,x3,a(13,1,getmid.c,pos,[]))).
27 block_condition(8,cond(supegal,x1,x3,a(10,1,getmid.c,neg,[]))).
28 block_condition(7,cond(inf,x1,x3,a(10,1,getmid.c,pos,[]))).
29 block_condition(5,cond(supegal,x1,x2,a(7,1,getmid.c,neg,[]))).
30 block_condition(4,cond(inf,x1,x2,a(7,1,getmid.c,pos,[]))).
31 block_condition(3,cond(supegal,x2,x3,a(5,1,getmid.c,neg,[]))).
32 block_condition(2,cond(inf,x2,x3,a(5,1,getmid.c,pos,[]))).
```

FIG. 7.16 – Termes générés à l'instrumentation de la fonction `getmid`

Ainsi, nous restreignons le domaine de chaque variable d'entrée à l'intervalle $[-100..100]$, nous avons donc :

$$Def(getmid)_{|user} = [-100..100] \times [-100..100] \times [-100..100]$$

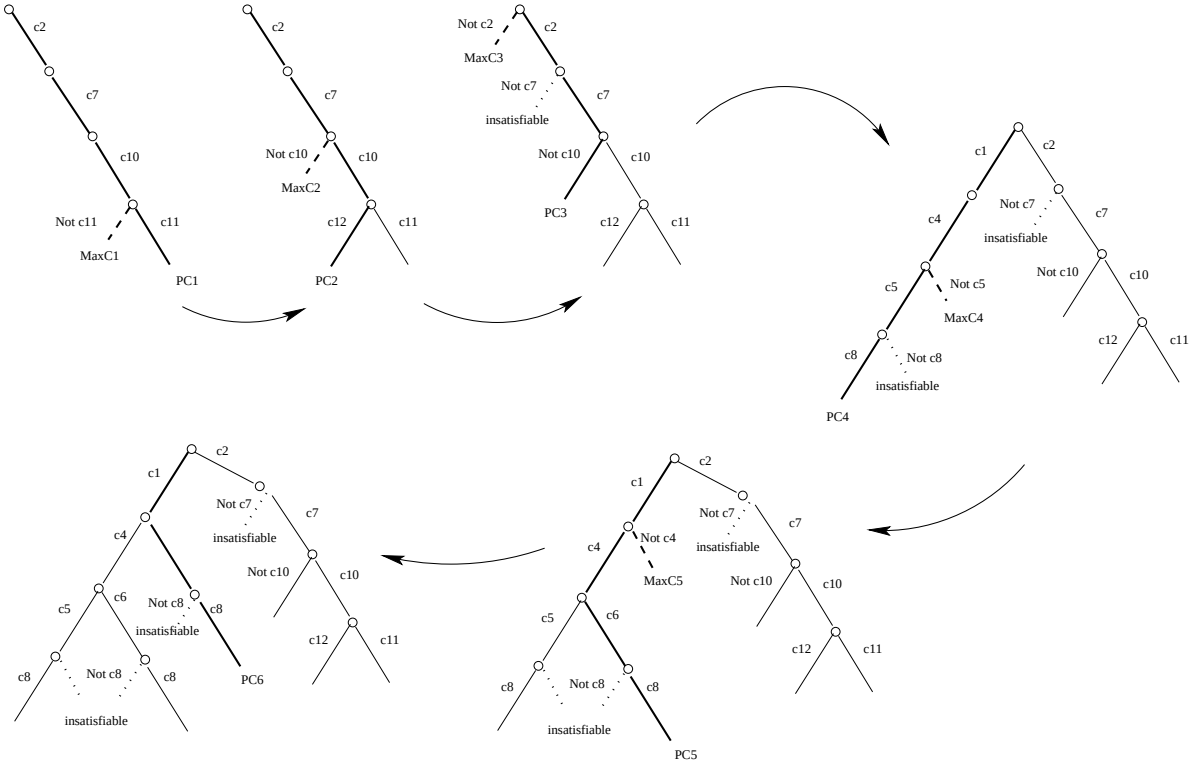

FIG. 7.17 – Illustration de notre stratégie sur la fonction `getmid`

Convention 7
Pour c_i une condition de branche, nous noterons c'_i son expression en fonction des valeurs initiales des variables d'entrées.

Le premier domaine de sélection SSD_0 correspond donc à $Def(getmid)_{|user}$ dans lequel un premier vecteur d'entrée X_1 est choisi aléatoirement tel que $X1 = (-21, 26, -31)$. Le prédicat de chemin associé est $PC(Ch_1, getmid, X) = (c'2 \wedge c'7 \wedge c'10 \wedge c'11)$. On applique la stratégie de sélection de cas de test pour déterminer $MaxC_1$ en conservant le plus long préfixe réversible de $PC(Ch_1, getmid, X)$ et en niant sa dernière contrainte. On obtient $MaxC_1(X) = (c'2 \wedge c'7 \wedge c'10 \wedge \neg c'11)$.

Le second vecteur d'entrée X_2 est une solution de $MaxC_1(X)$ tel que $X_2 = (30, 72, 58)$. Le prédicat associé est $PC(Ch_2, getmid, X) = (c'2 \wedge c'7 \wedge c'10 \wedge c'12)$ de la même façon on détermine $MaxC_2(X) = (c'2 \wedge c'7 \wedge \neg c'10)$.

Le troisième cas de test correspond à $X_3 = (13, -53, -73)$,
$PC(Ch_3, getmid, X) = (c'_2 \wedge c'_7 \wedge c'_9)$. Le plus long préfixe non exploré $c'_2 \wedge c'_7$ or $c'_2 \wedge \neg c'_7$ non réversible ici car c'_2 et c'_7 sont des contraintes équivalentes. Par backtrack, nous récupérons donc le second plus long préfixe réversible de $PC(Ch_3, getmid, X)$ à savoir $MaxC_3(X) = (\neg c'_2)$.

Pour le quatrième cas de test, nous obtenons $X_4 = (21, -12, 39)$,
$PC(Ch_4, getmid, X) = (c'_1 \wedge c'_4 \wedge c'_5 \wedge c'_8)$. Le système $c'_1 \wedge c'_4 \wedge c'_5 \wedge \neg c'_8$ est également insatisfiable $c'_8 = c'_1$ donc toujours par backtrack on obtient $MaxC_4(X) = (c'_1 \wedge c'_4 \wedge \neg c'_5)$.

Pour le cinquième cas de test, nous avons $X_5 = (17, -60, -11)$,
$PC(Ch_5, getmid, X) = (c'_1 \wedge c'_4 \wedge c'_6 \wedge c'_8)$ et $MaxC_5(X) = (c'_1 \wedge \neg c'_4)$ car $c'_1 \wedge c'_4 \wedge c'_6 \wedge \neg c'_8$ est insatisfiable pour les mêmes raisons que le test précédent.

Enfin le dernier cas de test correspond au vecteur d'entrée $X_6 = (21, 71, 85)$ avec $PC(Ch_5, getmid, X) = (c'_1 \wedge c'_3 \wedge c'_8)$, le système de contraintes $c'_1 \wedge c'_3 \wedge \neg c'_8$ étant insatisfiable toujours pour les mêmes raisons et tous les autres préfixes ayant déjà été explorés, le test est donc terminé pour cette fonction.

La figure 7.17 contient le sous-graphe des chemins faisables du CFG découverts le long des différents cas de test.

7.5.3 Analyse

Les 6 chemins faisables de la fonction `getmid` ont bien été couverts et quatre préfixes insatisfiables ont été coupés de l'arbre de recherche :
- $c'_2 \wedge \neg c'_7$
- $c'_1 \wedge c'_4 \wedge c'_5 \wedge \neg c'_8$
- $c'_1 \wedge c'_4 \wedge c'_6 \wedge \neg c'_8$
- $c'_1 \wedge c'_3 \wedge \neg c'_8$

Les 10 chemins insatisfiables ont été éliminés de notre étude par la coupure des branches associées aux préfixes insatisfiables identifiés.

Nous rappelons que notre méthode a effectué des choix aléatoires pour le premier cas de test dans $Def(getmid)_{|user}$ et pour les cas de test suivants dans les domaines de sélection successifs SSD_i. Ainsi, afin d'analyser et de valider les résultats de notre méthode, nous l'avons exécutée 10 fois sur la fonction `getmid`.

A chaque fois, 6 cas de test sont générés correspondant aux 6 chemins exécutables de la fonction et les 4 préfixes insatisfiables sont identifiés ce qui permet bien d'éliminer les 10 chemins infaisables.

Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.01, 0.02, 0.00, 0.02, 0.01, 0.01, 0.00, 0.03, 0.01, 0.01. Ces temps incluent l'exécution de la fonction instrumentée ainsi que les nombreuses opérations sur des fichiers (communication et trace). De plus, nous ne notons pas de variation notable dans les temps d'exécution. Ces résultats sont fortement encourageants de par leur stabilité et leur vitesse ce qui laisse encore entrevoir des améliorations pour l'avenir dans la mesure où notre implantation peut encore être optimisée.

Dans ce chapitre, nous nous sommes concentrés sur notre méthode de test unitaire structurelle PathCrawler pour laquelle nous avons présenté le principe de base, ses caractéristiques et les détails des différentes étapes. Dans la dernière section, nous l'avons illustrée en l'appliquant à la fonction `getmid`, fonction contenant de nombreux chemins infaisables.

Dans le chapitre suivant, nous allons présenter son application sur un exemple plus complet (structures répétitives, chemins infaisables, préconditions et tableaux) qui permettra d'illustrer plus en détails les points complexes traités par la méthode. Ce chapitre dressera également une analyse plus approfondie de PathCrawler et permettra d'introduire la troisième partie de ce manuscrit.

Chapitre 8

Illustration détaillée et analyse

Dans ce chapitre nous allons appliquer la méthode PathCrawler à la fonction `merge`, fonction de tri fusion de tableaux d'entiers. Cette fonction va nous permettre de démontrer plus en détails les points forts de notre méthode en ce qui concerne le calcul de prédicats de chemin, la manipulation de tableaux, la prise en compte de préconditions, l'application du critère des k-chemins et enfin la détection des chemins infaisables.

Ensuite, nous dresserons un bilan critique de notre approche et nous expliquerons pourquoi nous avons choisi de nous orienter vers une méthode de test mixte c'est-à-dire une méthode de test intégrant à la fois les aspects structurel et fonctionnel de la fonction sous test ce qui nous permettra d'introduire et de justifier la troisième partie de ce manuscrit.

8.1 Application de la méthode PathCrawler à une fonction de tri fusion

Dans cette section, nous allons utiliser l'exemple de la fonction merge dont le code source est présenté dans la figure 8.1.

8.1.1 Etude de la fonction `merge`

Cette fonction possède quatre variables d'entrée `t1[]`, `t2[]`, `l1`, `l2` respectivement deux tableaux d'entiers et les tailles des tableaux associées. Le tableau `t3[]` est l'unique variable de sortie de la fonction dans lequel tous les éléments des tableaux en entrée vont être insérés par ordre croissant.

Une grande partie des chemins d'exécution de la fonction sont en réalité infaisables. En effet, la première structure répétitive utilise tous les éléments de `t1` ou de `t2`, la seconde structure répétitive utilise les éléments restants de `t1` et la troisième structure répétitive utilise les derniers éléments de `t2`. Ainsi, la sortie de la première structure répétitive signifie qu'un des deux tableaux a été vidé et donc que seule une des deux structures répétitives suivantes sera exécutée.

La fonction `merge` est tout d'abord soumise à notre prétraitement des fonctions dont le résultat est contenu dans la figure 8.2. La figure 8.3 contient les clauses Prolog générées à l'instrumentation.

Dans la figure 8.3, nous pouvons observer en particulier les annotations de contraintes mises en place pour les structures répétitives. Par exemple, à la ligne 28 l'annotation de la clause $block\_condition\ a(5, 1, merge, neg, [b(pexit)])$ signifie qu'il s'agit de la première condition de la ligne 5 de la fonction initiale, que cette condition est niée ce qui correspond à la sortie de la structure répétitive (condition multiple conjonctive). De même, à la ligne 27, l'annotation

```
1  void merge (int t1[], int t2[], int t3[], int l1, int l2) {
2    int i = 0;
3    int j = 0;
4    int k = 0;
5    while (i < l1 && j < l2) {/*structure répétitive 1 */
6      if (t1[i] < t2[j]) { /*structure conditionnelle*/
7        t3[k] = t1[i];
8        i++;
9        }
10     else {
11       t3[k] = t2[j];
12       j++;
13       }
14     k++;
15     }
16   while (i < l1) {/*structure répétitive 2*/
17     t3[k] = t1[i];
18     i++;
19     k++;
20     }
21   while (j < l2) {/*structure répétitive 3*/
22     t3[k] = t2[j];
23     j++;
24     k++;
25     }
26 }
```

FIG. 8.1 – Fonction `merge`

$a(5, 2, merge, pos, [b(entry)])$ signifie qu'à la ligne 5, la seconde condition élémentaire est vérifiée ce qui correspond à l'entrée dans la structure répétitive.

8.1.2 Données fournies par l'utilisateur

La fonction possède 5 paramètres formels mais seulement 4 de ces paramètres sont de réelles variables d'entrées de la fonction. De plus, les tableaux `t1`, `t2` sont de taille variable. Nous définissons de façon automatique un sur-ensemble des variables d'entrée de la fonction dans lequel nous demandons à l'utilisateur de définir les réelles variables d'entrée de la fonction `merge`. Par défaut, les domaines associés aux variables d'entrée sont les domaines associés aux types C déclarés et le domaine des dimensions des structures de données est $[1, 1]$ mais nous offrons à l'utilisateur la possibilité de définir son propre domaine en entrée pour la fonction $Dom(merge)_{|user}$ qui correspond à un domaine restreint du domaine global de la fonction.

Dans le cas de variables d'entrée structurées, l'utilisateur doit définir chaque domaine des composants de ces variables comme les éléments d'un tableau, les champs d'une structure, les valeurs déréférencées,.... Une forme limitée de la quantification universelle (c'est-à-dire sur des domaines finis) peut être utilisée pour définir les domaines des composants indexés.

Nous demandons également à l'utilisateur de fournir les relations de dépendances entre les variables et les contraintes associées définissant les conditions de bonne utilisation de la fonction. Dans notre exemple, la variable d'entrée `l1` est utilisée comme indice pour accéder aux éléments de `t1` de même pour `l2` et `t2` ainsi les variables `l1, l2` ne peuvent être de valeurs négatives et ne peuvent excéder la valeur de la taille des tableaux associés. De plus, les éléments de `t1` et de `t2` doivent être initialement triés par ordre croissant.

Les propriétés fournies par l'utilisateur pour notre exemple sont donc :

```
1  void merge(int *t1 , int *t2 , int *t3 , int l1,int l2 )
2  { int i ;  int j ;  int k ;
3    {
4    printf(":1");
5    i = 0; j = 0; k = 0;{
6    while (1) {
7      while_0_continue:;
8      if (i < l1) {                    /*c1*/
9        printf(".2");
10       if (j < l2) {                  /*c3*/
11         printf(".4");
12       } else {                       /*c4*/
13         printf(".5");
14         goto while_0_break;}}
15   } else {                           /*c2*/
16     printf(".3");
17     goto while_0_break;}
18     if ((*(t1 + i)) < (*(t2 + j))) { /*c5*/
19       printf(".6:8");
20       (*(t3 + k)) = (*(t1 + i));
21       i ++;
22     } else {                         /*c6*/
23       printf(".7:9");
24       (*(t3 + k)) = (*(t2 + j));
25       j ++;}
26     printf(":10");
27     k ++;}
28   while_0_break:;}
29   { while (1) {
30     while_1_continue: ;
31     if (i < l1) {                    /*c7*/
32       printf(".11");
33     } else {                         /*c8*/
34       printf(".12");
35       goto while_1_break;}
36     printf(":13");
37     (*(t3 + k)) = (*(t1 + i));
38     i ++;k ++;}
39   while_1_break: ;}
40   { while (1) {
41     while_2_continue:;
42     if (j < l2) {                    /*c9*/
43       printf(".14");
44     } else {                         /*c10*/
45       printf(".15");
46       goto while_2_break; }
47     printf(":16");
48     (*(t3 + k)) = (*(t2 + j));
49     j ++;k ++; }
50   while_2_break:;}
51   return;
52 }}
```

FIG. 8.2 – Fonction `merge` prétraitée et instrumentée

```
block_assignments(16,[affect(cont(t3, k), cont(t2, j)),affect(j, +(j, 1)),
affect(k, +(k, 1))]).
block_assignments(15,[]).
block_assignments(14,[]).
block_assignments(13,[affect(cont(t3, k), cont(t1, i)),affect(i, +(i, 1)),
affect(k, +(k, 1))]).
block_assignments(12,[]).
block_assignments(11,[]).
block_assignments(10,[affect(k, +(k, 1))]).
block_assignments(9,[affect(cont(t3, k), cont(t2, j)),affect(j, +(j, 1))]).
block_assignments(8,[affect(cont(t3, k), cont(t1, i)),affect(i, +(i, 1))]).
block_assignments(7,[]).
block_assignments(6,[]).
block_assignments(5,[]).
block_assignments(4,[]).
block_assignments(3,[]).
block_assignments(2,[]).
block_assignments(1,[affect(i, 0),affect(j, 0),affect(k, 0)]).
block_assignments(0,[]).
block_condition(15,cond(supegal,j,l2,a(21,1,merge,neg,[b(exit)]))).
block_condition(14,cond(inf,j,l2,a(21,1,merge,pos,[b(entry)]))).
block_condition(12,cond(supegal,i,l1,a(16,1,merge,neg,[b(exit)]))).
block_condition(11,cond(inf,i,l1,a(16,1,merge,pos,[b(entry)]))).
block_condition(7,cond(supegal,cont(t1, i),cont(t2, j),a(6,1,merge,neg,[]))).
block_condition(6,cond(inf,cont(t1, i),cont(t2, j),a(6,1,merge,pos,[]))).
block_condition(5,cond(supegal,j,l2,a(5,2,merge,neg,[b(exit)]))).
block_condition(4,cond(inf,j,l2,a(5,2,merge,pos,[b(entry)]))).
block_condition(3,cond(supegal,i,l1,a(5,1,merge,neg,[b(pexit)]))).
block_condition(2,cond(inf,i,l1,a(5,1,merge,pos,[b(pentry)]))).
```

FIG. 8.3 – Clauses Prolog générées pour la fonction `merge`

- $dim(t1) = l1$
- $dim(t2) = l2$
- $l1 \in [0..100]$
- $l2 \in [0..100]$
- $\forall i \in [0..l1-1], t1[i] \in [0..100]$
- $foralli \in [1..l1-1], t1[i] \geq t1[i-1]$
- $\forall j \in [0..l2-1], t2[j] \in [0..100]$
- $\forall j \in [0..l2-1], t2[j] \geq t1[j-1]$

De plus, nous fixons la valeur de k à 2 dans notre exemple pour l'application du critère des k-chemins ce qui correspond à 126 chemins théoriques dont seuls 17 sont en réalité infaisables.

Enfin, l'utilisateur peut également fournir un oracle sous la forme d'une fonction C prenant en arguments à la fois les variables d'entrée et les variables de sortie de la fonction `merge` afin de pouvoir établir un verdict de test à chaque cas de test.

8.1.3 Cas de test successifs

Le premier vecteur d'entrée, X_1, est choisi dans le domaine restreint par l'utilisateur de `merge` respectant les différentes contraintes fournies par l'utilisateur `merge` soit

$$X_1 \in SD_0$$

$$SD_O = Def(merge)_{|user}$$

Nous donnerons les chemins suivis par les différents cas de tests en terme de séquence de conditions de branches vérifiées selon les annotations de 8.2.

Ainsi, le premier vecteur d'entrée est $X_1 = ([], [], 0, 0)$ qui correspond à la séquence $Ch1 = (c2, c8, c10)$ et au prédicat de chemin $PC(Ch_1, merge, X) = (0 \geq l1) \wedge (0 \geq l1) \wedge (0 \geq l2)$. Le plus long préfixe réversible non exploré est $PC(Ch_1, merge, X)$ ainsi $MaxC_1 = (0 \geq l1) \wedge (0 \geq l1) \wedge \neg(0 \geq l2)$.

Pour le second vecteur d'entrée, solution de $MaxC_1$, `t2` possède cette fois un élément et `t1` est toujours vide. Il n'y a donc qu'un seul passage dans la troisième structure répétitive de la fonction.

Le troisième cas de test, correspondra à `t1` vide et `t2` à deux éléments soit à deux passages dans la troisième structure répétitive. La détermination de $MaxC_3(X)$ prendra en compte le nombre d'itérations dans cette troisième structure en évitant un troisième passage ($k = 2$) dans la structure répétitive ce qui donnera par backtrack $MaxC_3(X) = \neg(0 \geq l1)$ (cf. figure 8.4) et ainsi de suite pour les cas de test suivants.

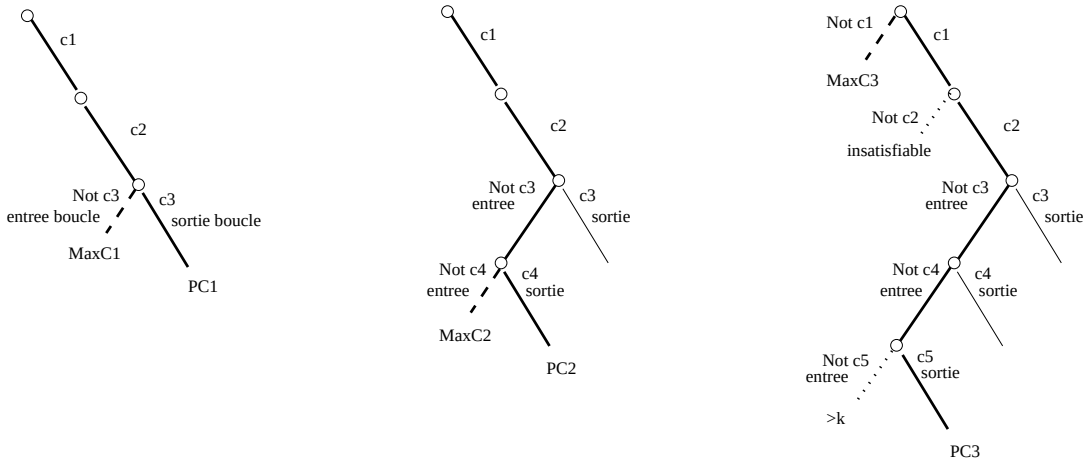

FIG. 8.4 – Illustration de notre stratégie sur la fonction `merge`

Nous donnons ci-dessous la liste des cas de test effectués sur la fonction merge avec pour convention les valeurs des éléments de tableaux sous forme d'une liste comme composant de X_i et ci' l'expression de la condition ci après substitution en termes des valeurs initiales des variables d'entrée. De plus, nous notons $TestMaxC_i$ la première tentative pour calculer $MaxC_i$ dans la mesure où parfois $TestMaxC_i$ sera insatisfiable ou possédera plus de k itérations par structure répétitive.

1. $X_1 = ([], [], 0, 0)$, $Ch_1 = (c2, c8, c10)$,
   $TestMaxC_1 = (c2' \wedge c8' \wedge \neg c10')$
2. $X_2 = ([], [-3], 0, 1)$, $Ch_2 = (c2, c8, c9, c10)$,
   $TestMaxC_2 = (c2' \wedge c8' \wedge \neg c9')$
3. $X_3 = ([], [-52, 30], 0, 2)$, $Ch_3 = (c2, c8, c9, c9, c10)$,
   $TestMaxC_3 = (c2' \wedge c8' \wedge c9' \wedge \neg c9')$
4. $X_4 = ([-5], [], 1, 0)$, $Ch_4 = (c1, c4, c7, c8, c10)$,
   $TestMaxC_1 = (\neg c1')$
5. $X_5 = ([-41, -8], [], 2, 0)$, $Ch_5 = (c1, c4, c7, c7, c8, c10)$,
   $TestMaxC_5 = (c1' \wedge c4' \wedge c7' \wedge \neg c7')$
6. $X_6 = ([-17], [16], 1, 1)$, $Ch_6 = (c1, c3, c5, c2, c8, c9, c10)$,
   $TestMaxC_6 = (c1' \wedge \neg c3')$

7. $X_7 = ([24], [67, 88], 1, 2)$, $Ch_7 = (c1, c3, c5, c2, c8, c9, c9, c10)$,
$TestMaxC_7 = (c1' \wedge c3' \wedge c5' \wedge c2' \wedge c8' \wedge c9' \wedge \neg c9')$
8. $X_8 = ([-67, 14], [-22], 2, 1)$, $Ch_8 = (c1, c3, c5, c1, c3, c6, c1, c4, c7, c8, c10)$,
$TestMaxC_8 = (c1' \wedge c3' \wedge c5' \wedge \neg c1')$
9. $X_9 = ([-77, -27, 0], [-61], 3, 1)$, $Ch_9 = (c1, c3, c5, c1, c3, c6, c1, c4, c7, c7, c8, c10)$,
$TestMaxC_9 = (c1' \wedge c3' \wedge c5' \wedge c1' \wedge c3' \wedge c6' \wedge c1' \wedge c4' \wedge c7' \wedge \neg c7')$
10. $X_{10} = ([-1, 23], [46], 2, 1)$, $Ch_{10} = (c1, c3, c5, c1, c3, c5, c2, c8, c9, c10)$,
$TestMaxC_{10} = (c1' \wedge c3' \wedge c5' \wedge c1' \wedge c3' \wedge \neg c5')$
11. $X_{11} = ([-68, -37], [-14, 29], 2, 2)$, $Ch_{11} = (c1, c3, c5, c1, c3, c5, c2, c8, c9, c9, c10)$,
$TestMaxC_{11} = (c1' \wedge c3' \wedge c5' \wedge c1' \wedge c3' \wedge c5' \wedge c2' \wedge c8' \wedge c9 \wedge \neg c9')$
12. $X_{12} = ([-69, -36, 28], [-5], 3, 1), Ch_{12} = (c1, c3, c5, c1, c3, c5, c1, c3, c6, c1, c4, c7, c7, c7, c8, c10)$,
$TestMaxC_{12} = (c1' \wedge c3' \wedge c5' \wedge c1' \wedge \neg c'3)$
13. $X_{13} = ([-23], [-50], 1, 1)$, $Ch_{13} = (c1, c3, c6, c1, c4, c7, c8, c10)$,
$TestMaxC_{13} = (c1' \wedge c3' \wedge \neg c6')$
14. $X_{14} = ([41, 73], [9], 2, 1)$, $Ch_{14} = (c1, c3, c6, c1, c4, c7, c7, c8, c10)$,
$TestMaxC_{14} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge c4' \wedge c7' \wedge \neg c7')$
15. $X_{15} = ([-30], [-69, 24], 1, 2)$, $Ch_{15} = (c1, c3, c6, c1, c3, c5, c2, c8, c9, c10)$,
$TestMaxC_{15} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge \neg c3')$
16. $X_{16} = ([-30], [-73, -13, 15], 1, 3)$, $Ch_{16} = (c1, c3, c6, c1, c3, c5, c2, c8, c9, c9, c10)$,
$TestMaxC_{16} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge c3' \wedge c5' \wedge c2' \wedge c8' \wedge c9' \wedge \neg c9')$
17. $X_{17} = ([31, 56], [-17, 64], 2, 2)$, $Ch_{17} = (c1, c3, c6, c1, c3, c5, c1, c3, c5, c2, c8, c9, c10)$,
$TestMaxC_{17} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge c3' \wedge c5' \wedge \neg c1')$
18. $X_{18} = ([27], [-54, -26], 1, 2)$, $Ch_{18} = (c1, c3, c6, c1, c3, c6, c1, c4, c7, c8, c10)$,
$TestMaxC_{18} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge c3' \wedge \neg c6')$
19. $X_{19} = ([-52, -26], [-79, -65], 2, 2)$, $Ch_{19} = (c1, c3, c6, c1, c3, c6, c1, c4, c7, c7, c8, c10)$,
$TestMaxC_{19} = (c1' \wedge c3' \wedge c6' \wedge c1' \wedge c3' \wedge c6' \wedge c1' \wedge c4' \wedge c7' \wedge \neg c7')$

Nous avons donc généré 19 cas de test différents pour 17 chemins exécutables : les cas de test 12 et 17 contiennent plus de 2 itérations par structure répétitive : nous rappelons que nous ne garantissons pas la non génération de cas de test superflus comme expliqué dans la section 7.3.2. Nous avons identifié en tout 25 préfixes de chemins insatisfiables ou $>k$ éliminant 109 chemins de la fonction.

Pour prouver l'efficacité de la méthode, nous exécutons PathCrawler 10 fois de suite sur la fonction `merge` avec $k = 5$ et des domaines non restreints pour les éléments des tableaux en entrée. Le nombre total de chemins est de 4536 dont seuls 321 sont faisables et respectent k. A chaque exécution, 337 cas de test sont générés et 317 préfixes insatisfiables ou $>$ k sont éliminés de l'arbre de recherche excluant 4215 chemins.

Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont :2.06, 2.08, 2.04, 2.05, 1.98, 2.09, 1.96, 2.04, 2.06, 2.02.

Les résultats sont prometteurs et notre heuristique de labelling parait efficace car en 2 secondes 654 préfixes prédicats ont été générés ou rejetés, ce temps incluant l'exécution de la fonction et les diverses communications et opérations entre fichiers.

8.2 Analyse de la méthode PathCrawler

8.2.1 Points en étude

Comme nous l'avons précisé dans le chapitre 7, nous traitons un sous-ensemble du langage C ANSI mais certains points non traités sont actuellement en étude (cf. section 7.2.1).

Nous ne traitons par encore, par exemple, les codes récursifs à cause du problème combinatoire de chemins. Cependant, une limitation du nombre d'appels de ces fonctions selon le même principe que le traitement des structures répétitives à nombre variable d'itérations est envisagé. Un traitement dynamique des appels de fonctions récursives lors de l'instrumentation permettra la mise en correspondance des variables formelles de la fonction appelée aux variables affectives de l'appel en terme des variables d'entrée.

Notre méthode de substitution offre un traitement complet des alias y compris pour les éventuelles relations sur les variables d'entrée influant sur le chemin emprunté. Cependant, un cas n'est pas traité : quand une variable en entrée pointe vers une autre variable en entrée.

La résolution de contraintes en présence de flottants définie par Gotlieb, Michel et Botella [BGM06] n'est pas encore intégrée dans notre approche. Pour l'instant, nous utilisons les réels d'Eclipse [WNJ97] en attendant d'implanter les résultats de travaux précédemment cités.

Une de nos autres perspectives est, comme nous l'avons déjà laisser entendre, d'étendre notre méthode à d'autres langages de programmation impératifs afin ne plus se limiter à la seule étude du langage C.

De plus, notons que le test aux bords des domaines est un test plus fort que le test des chemins car, statistiquement, les défauts se situent plus souvent aux limites des domaines des valeurs d'entrée [ABC+02].

8.2.2 Étapes non automatisées

Notre méthode s'adresse aux langages impératifs et plus particulièrement au langage C pour lequel un prototype a été implanté. Cependant, certains points demandent toujours l'intervention de l'utilisateur et restent donc manuels.

En effet, l'utilisateur, dans la configuration actuelle, doit identifier tout d'abord les entrées et sorties effectives de la fonction (y compris les éventuelles variables globales) dont l'outil fournit un sur-ensemble. De plus, il doit pouvoir préciser certains types : par exemple, les tableaux de taille variable sont considérés comme des pointeurs. Il faut donc que l'utilisateur précise ce point en modifiant le type de la variable identifiée comme un pointeur en un tableau de taille variable et si possible de fournir une variable contenant la taille de ce tableau (comme pour l'exemple de la fonction `merge` où la variable `l1` contient la taille du tableau de taille variable `t1`). L'utilisateur peut fournir ces différentes données nécessaires d'une façon simple via une interface intuitive.

Ainsi, l'utilisateur doit parfois préciser les définitions des domaines des variables d'entrée et des contraintes d'environnement associées. Il doit également fournir les préconditions de la fonction nécessaires pour déterminer le domaine de définition de la fonction.

Pour l'établissement d'un verdict de test, l'utilisateur devra également fournir une fonction C jouant le rôle d'oracle pour la fonction sous test.

8.2.3 Intégration d'aspects fonctionnels à la méthode PathCrawler

Le test structurel est une stratégie nécessaire dans la mesure où elle seule peut détecter certaines erreurs d'implantation. Le but est de s'assurer que la totalité des différentes exécutions possibles du logiciel a bien été explorée. Cependant, les spécifications deviennent nécessaires pour la mise en œuvre d'un oracle qui analyse la correction des sorties du programme ou pour le traitement du problème des chemins manquants [GG75].

L'idée du test fonctionnel est, quant à lui, de vérifier le comportement réel du logiciel par rapport à son comportement spécifié. Son apport principal est justement l'élimination du problème des chemins manquants et de permettre la mise en place d'un verdict de test.

Nous avons envisagé, hormis l'amélioration des aspects manquants de notre approche, une extension de notre méthode vers d'autres contextes afin de ne plus se limiter simplement à une technique de test structurel. Ces perspectives sont possibles grâce à la faible complexité et à l'efficacité de notre méthode de test. L'objectif de la stratégie de test mixte est d'atteindre un test complet du point de vue de l'implantation et des spécifications.

Cette extension est possible en enrichissant l'instrumentation de la fonction par des informations issues des spécifications et par le fait que notre stratégie de test reste ouverte à l'ajout de nouvelles contraintes.

Prenons par exemple le problème des chemins manquants adressé uniquement par des méthodes fonctionnelles. Pour que notre méthode de test adresse également ce point et en supposant disposer des spécification complètes de la fonction sous test exprimées via un langage de spécification sous forme de couples pre/post 3, il s'agirait de couvrir structurellement chaque domaine des couples pre/post c'est-à-dire chaque domaine fonctionnel DF_i de la fonction sous test.

De plus, l'analyse automatisée des préconditions formalisées dans la spécification de la fonction nous permettrait de générer automatiquement le domaine de définition de la fonction en excluant du domaine les valeurs interdites, en identifiant clairement les variables d'entrée et de sortie de la fonction et en prenant en compte les contraintes entre les variables au lieu de le demander à l'utilisateur.

Enfin, pour le problème de la génération automatique d'un oracle de test, il s'agirait de couvrir structurellement chaque domaine fonctionnel, DF_i, de la fonction en utilisant la postcondition $Q_i(f, X, Y)$[1] comme oracle pour ce domaine fonctionnel.

8.2.4 Traitement des appels de fonctions dans la fonction sous test

Le traitement des appels de fonction dans la fonction sous test consiste, dans PathCrawler, à étendre le critère de test au code source des fonctions appelées. Il s'agit d'un traitement "inlining" des fonctions appelées. L'avantage d'un tel traitement est de prendre en compte le comportement réel des fonctions appelées. Le problème associé à ce traitement est le même que le problème de la gestion des structures répétitives du code : la combinatoire des chemins des fonctions appelées s'ajoute à celle de la fonction sous test rendant l'application d'un critère stricte comme celui des k-chemins inapplicable pour des fonctions réalises avec appels de fonction.

[1] Nous rappelons qu'une postcondition $Q_i(f, X, Y)$ est un ensemble de contraintes caractérisant la relation entre les variables d'entrée et de sortie de la fonction pour un domaine fonctionnel donné DF_i (cf. chapitre 3).

Dans la partie suivante, nous allons expliquer la stratégie de traitement des fonctions appelées proposée dans ce manuscrit. Nous expliquerons également comme nous allons incorporer cette gestion des appels de fonction à la méthode PathCrawler que nous venons de présenter.

Dans cette partie, nous nous sommes intéressés au problème de la génération automatique de cas de test structurels et nous avons présenté quelques travaux associés.

Ensuite, nous nous sommes penchés sur notre méthode de génération de cas de test unitaire structurels, la méthode PathCrawler, dont nous avons expliqué les caractéristiques et les différentes étapes. Nous avons ensuite appliqué PathCrawler à deux fonctions exemples pour illustrer de façon plus concrète son déroulement.

Dans le dernier chapitre, nous avons fait un bilan critique de la méthode en expliquant ses limitations actuelles mais aussi les limitations propres à toutes méthodes purement structurelles. De ce constat, nous avons eu l'idée de modifier cette méthode pour y intégrer des aspects fonctionnels afin d'obtenir, comme nous le définissons, une méthode de test mixte.

La mise en place d'une stratégie de test mixte a pour objectif premier le traitement des appels de fonction traités par "inlining" initialement dans la méthode PathCrawler. Ce traitement "inlining" n'est pas adapté à une méthode de test structurel en général et en particulier pour un critère de test aussi rigoureux que celui que nous appliquons.

La partie suivante va nous permettre de présenter les traitements classiques des appels de fonction dans une méthode de test unitaire. Nous présenterons ensuite comment nous proposons d'abstraire les fonctions appelées via l'utilisation de leurs spécifications exprimées sous forme de couples pre/post. Enfin, nous utiliserons cette modélisation fonctionnelle des fonctions appelées dans la fonction sous test que nous proposons d'intégrer dans la méthode PathCrawler pour garantir le maintien de la couverture de la fonction sous test en limitant au maximum l'exploration des fonctions appelées.

Troisième partie

Génération de cas de test structurels et traitement des appels de fonctions

Cette partie présente la stratégie proposée pour la gestion des appels de fonction pour la méthode PathCrawler. Cette stratégie possède deux objectifs principaux à savoir le maintien de 100% des k-chemins de la fonction sous test et la limitation au maximum de l'exploration des chemins des fonctions imbriquées.

Un premier chapitre sera consacré au contexte général de test en termes de notions de contexte d'appel et de domaine d'appel des fonctions imbriquées et du graphe d'appel de la fonction sous test. Nous donnerons nos objectifs et motivations et le principe général de la stratégie de gestion des appels de fonctions proposée dans ce document.

Dans le second chapitre, nous dresserons un panorama des techniques de test d'intégration et de test imbriqué existantes. Nous verrons au cours de cette partie que notre gestion des appels de fonction se situe à la frontière de ces deux techniques de test.

Dans le chapitre suivant, nous décrirons plus en détails la stratégie de gestion des appels de fonction proposée dans ce manuscrit. Nous expliquerons l'abstraction des fonctions imbriquées par l'utilisation de leurs spécifications axiomatiques sous forme de couples pre/post comme expliqué dans le chapitre 3. Cette abstraction sera incluse dans le calcul des prédicats de chemins que nous désignerons comme prédicats de chemins mixtes dans la mesure où ceux-ci seront construits d'une part à partir des informations structurelles de la fonction sous test et fonctionnelles des fonctions imbriquées. Cette modélisation des prédicats de chemins nous permettra ainsi de modifier la notion de graphe de flot de contrôle d'une fonction contenant des blocs d'appel pour la construction de ce que nous désignons comme un graphe mixte. Le graphe mixte de la fonction sous test correspondant au CFG de la fonction sous test dans lequel les blocs d'appels ont été remplacés par une représentation fonctionnelle des fonctions imbriquées.

Cela nous amènera ensuite à soumettre notre modélisation des fonctions sous test contenant des instructions d'appels à la méthode PathCrawler selon un des premiers critères de test que nous définirons. Nous verrons que ce critère permet de maintenir la couverture structurelle des k-chemins de la fonction sous test tout en limitant l'exploration des fonctions imbriquées. Nous proposerons ensuite un autre critère de test à appliquer à la nouvelle modélisation des fonctions sous test avec instructions d'appel limitant la possible redondance des cas de test effectués dans la couverture des chemins structurels de la fonction sous test.

Un chapitre sera ensuite consacré à la validation de la méthode pour la soumission de la nouvelle modélisation des fonctions sous test avec instructions d'appels selon les deux nouveaux critères de test définis. Nous comparerons les résultats ainsi obtenus sur ces fonctions aux traitements habituels proposés pour la gestion des appels de fonction dans les méthodes de test unitaires par l'utilisation de bouchons fonctionnels et structurels.

Enfin, dans un dernier chapitre, nous dresserons un bilan de la stratégie. Nous analyserons en détails notre méthode en terme de couverture et nous dresserons une critique détaillée de notre stratégie. Pour finir, nous discuterons des extensions envisagées et envisageables de notre approche.

Chapitre 9

Notre approche

L'approche de PathCrawler, telle que présentée dans le chapitre 7, quant à la gestion des appels de fonctions consiste en un traitement dit "inlining" des fonctions imbriquées ce qui signifie que le critère de couverture de la fonction sous test est étendu aux fonctions qu'elle appelle. Un tel traitement ajoute la combinatoire des chemins des fonctions imbriquées à celle de la fonction sous test. Ainsi, pour une fonction sous test "non jouet" faisant appel à d'autres fonctions, l'application d'un tel critère structurel de test risque de devenir trop exigeant en terme de nombre de tests.

La problématique qui se pose est donc la suivante : *comment pouvons-nous limiter l'exploration des fonctions imbriquées et par conséquent le nombre de tests à exécuter tout en garantissant le maintien de 100% des (k-)chemins de la fonction sous test ?*

Dans ce chapitre, nous justifions notre approche et nous définissons le matériel nécessaire pour la suite à savoir la notion de contextes d'appel et domaines d'appel d'une fonction imbriquée et de graphe d'appel de la fonction appelante.

9.1 Motivation

L'objectif de notre approche est de proposer une approche plus efficace que le traitement "inlining" des fonctions imbriquées en terme d'explosion combinatoire des chemins tout en évitant de couvrir des chemins non exécutables en réalité pour ne pas émettre de fausses alertes.

Nous allons voir plus en détails l'incidence d'un traitement "inlining" des fonctions imbriquées sur la combinatoire des chemins de la fonction sous test.

9.1.1 Limitations du traitement "inlining" des fonctions imbriquées

Le traitement "inlining" consiste à **déplier le code des fonctions imbriquées**. Les instructions d'appels sont remplacées par le code source des fonctions imbriquées en adaptant le nom des différentes variables (paramètres, alias, etc).

Prenons la fonction `f` telle que $f(X) = Y$ de la figure 9.1 possédant un bloc d'appel pour la fonction imbriquée `g`. Il existe $n \geq 1$ chemins exécutables partiels dans la fonction `f` précédant l'appel de `g`.

Il existe $m \geq 1$ chemins partiels exécutables dans `f` succédant l'appel de `g`. Si nous nous limitons à la couverture structurelle de la fonction `f`, $n * m$ chemins contenant tous l'appel de la fonction imbriquée `g` sont à couvrir.

De plus, il existe dans le graphe de la fonction `g` imbriquée $p \geq 1$ chemins internes exécutables. Si nous déplions le code de la fonction imbriquée, l'appel de `g` est remplacé par son graphe de flot de contrôle en adaptant les noms des variables de la fonction imbriquée.

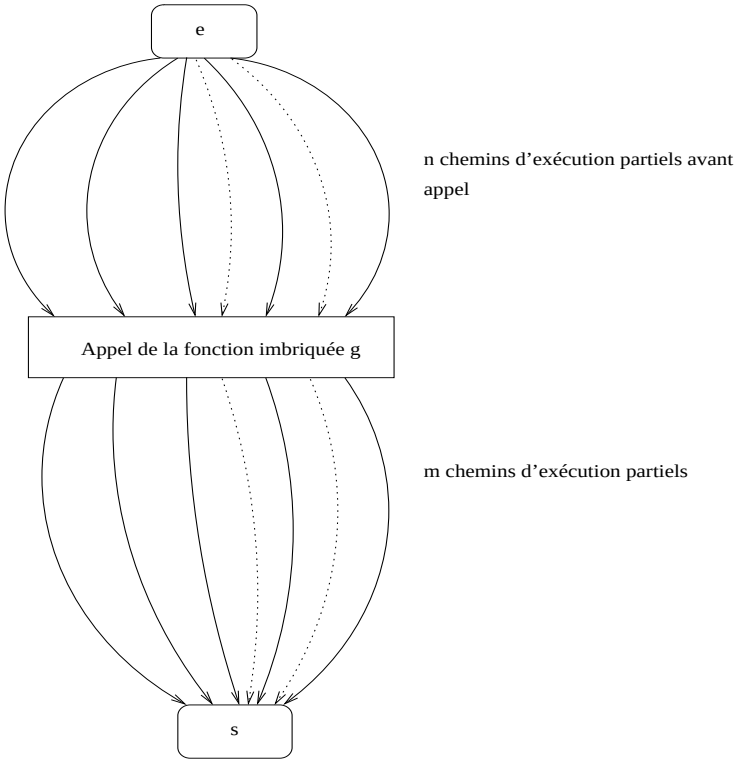

FIG. 9.1 – Graphe de contrôle de `f` schématisé

La figure 9.2 contient une schématisation du graphe de `f` de la figure 9.1 avec le dépliage de la fonction imbriquée contenant $n * m * p$ chemins au total. Cela correspond à un facteur p supplémentaire (p étant le nombre de chemins d'exécution faisables de la fonction imbriquée). Le dépliage de la fonction imbriquée introduit potentiellement des chemins infaisables supplémentaires à identifier.

Pour une fonction imbriquée assez simple et par conséquent contenant peu de chemins exécutables, cette stratégie de dépliage de la fonction imbriquée peut être une stratégie suffisante. En revanche, dès que la fonction imbriquée se complexifie (nombre conséquent de chemins, présence de structures répétitives, ...) ou dès lors que le nombre d'appels à des fonctions imbriquées augmente de façon significative, cette stratégie se heurte au problème de l'explosion combinatoire des chemins et par conséquent au nombre des cas de test à effectuer.

Ainsi, même si elle offre une bonne couverture de la fonction sans amener à couvrir des chemins inexécutables en réalité, le problème de l'explosion combinatoire des chemins est amplifié par cette technique.

9.1.2 Vers une stratégie de test mixte

DÉFINITION – 9.1.1

> *Une **méthode de test mixte** est une méthode de test utilisant à la fois des aspects fonctionnel et structurel pour la détermination des différents cas de test.*

Comme nous l'avons dit à la fin de la dernière partie, l'utilisation d'aspects fonctionnels dans une méthode de test structurelle permet d'envisager une amélioration de l'efficacité et une extension de la méthode de test.

Le test fonctionnel possède la capacité de tester une fonction complexe, au moins en partie, grâce à la possibilité de choisir le niveau de détail de la description fonctionnelle. Ainsi, il peut

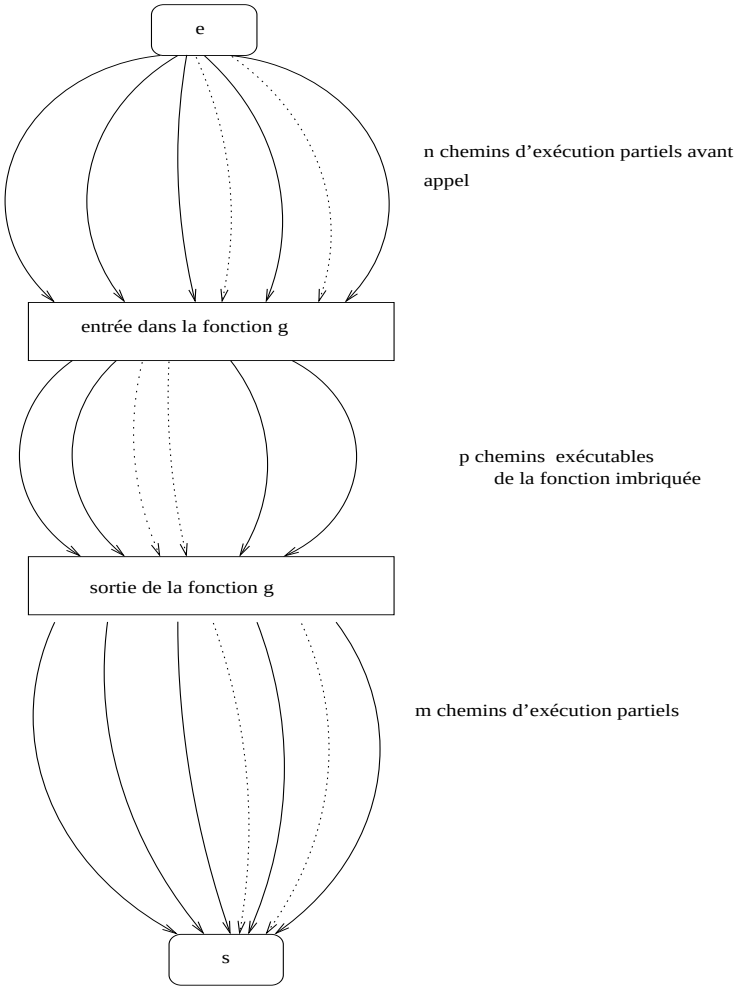

FIG. 9.2 – Graphe de contrôle de `f` avec dépliage de g

être appliqué à un code complexe (composé d'un très grand nombre de chemins) pour lequel une stratégie de test structurel poserait des problèmes d'efficacité et de combinatoire des chemins.

L'intérêt du test mixte consiste à utiliser le test fonctionnel pour pallier aux inconvénients du test structurel. Le test fonctionnel permet de choisir le niveau d'abstraction utilisé par le choix du niveau de détails de la spécification. Nous choisissons ainsi de limiter l'exploration des fonctions imbriquées en les abstrayant par une modélisation fonctionnelle reposant sur leurs spécifications.

Notre volonté de mise en place d'une technique de test mixte concerne particulièrement le problème de l'explosion combinatoire des chemins provoqué par un traitement "inlining"[1] des fonctions imbriquées dans la fonction sous test.

Notre idée est donc de remplacer le traitement "inlining" des fonctions imbriquées par une stratégie mixte de gestion des appels de fonctions.

[1] Rappelons que cela signifie que le critère de test des k-chemins est appliqué également au code source des fonctions imbriquées dans la fonction sous test.

9.2 Contextes d'appel d'une fonction

L'exécution d'une fonction imbriquée est dépendante de son environnement c'est-à-dire pour notre cas à l'exécution de la fonction appelante.

Dans une instruction d'appel, la configuration d'appel établie par les paramètres effectifs peut influencer directement le comportement de la fonction imbriquée.

De plus, les chemins partiels de la fonction appelante précédant une instruction d'appel influent directement sur le comportement d'une fonction imbriquée. En effet, le domaine en entrée d'une fonction imbriquée est restreint par les calculs et prédicats de chemin associés à ces chemins partiels qui réduisent les domaines des expressions effectives d'appel, des variables globales et des variables référencées par les paramètres effectifs d'appel utilisées comme entrées de la fonction imbriquée.

Illustration 63
Soient une fonction imbriquée `g` *de profil :* $g : int \times int \times int \rightarrow int$, $\mathcal{P}_{form} = [a, b]$ *la liste des paramètres formels de* `g` *où* a *et* b *sont de type* int, *la liste des paramètres effectifs* $\mathcal{P}_{eff} = [x, y - x]$ *où* x *et* y *sont aussi de type* int, *un chemin partiel précédant l'instruction d'appel est* $Ch_p : (x > 15, y = x + 3, v = y + x)$ *où* v *est une variable globale de la fonction appelante. Les deux premières entrées de la fonction* `g` *correspondent aux paramètres formels et la troisième à la variable globale utilisée en entrée. Après l'exécution du chemin partiel, nous pouvons réduire les domaines des variables de la fonction sous test :*

- $x \in [16, MaxInt]$
- $y \in [19, MaxInt]$ *et*
- $v \in [35, MaxInt]$

Pour ce chemin partiel précédant l'appel et par la configuration d'appel des paramètres effectifs, la fonction imbriquée est appelée sur sont domaine d'entrée restreint à :

$$Dom(x)_{|CoDom(Ch_p)} \times Dom(y - x)_{|CoDom(Ch_p)} \times Dom(v)_{|CoDom(Ch_p)} =$$
$$[16, MaxInt] \times ([MinInt, MaxInt - 16]) \times [35, MaxInt]$$

La notion de contexte d'appel correspond à une exécution donnée d'une fonction imbriquée caractérisée par le couple composé du chemin partiel précédant l'appel et par la liste des expressions effectives d'appel.

9.2.1 Définition d'un contexte d'appel

Définition – 9.2.1

> *Un* ***contexte d'appel*** *d'une fonction est caractérisé par le couple* $(Ch_{p_i}, \mathcal{P}_{eff_i})$ *où* Ch_{p_i} *est le chemin précédant l'appel imbriqué dans la fonction appelante et* $\mathcal{P}_{eff_i}$ *est la liste des paramètres effectifs de l'appelant pour le chemin associé.*

A un bloc d'appel donné d'une fonction peut correspondre différents chemins partiels et donc différents contextes d'appels. Une fonction imbriquée est caractérisée par l'ensemble de ses contextes d'appel.

Définition – 9.2.2

> *L'****ensemble des contextes d'appel*** *d'une fonction imbriquée* `g` *noté* $CtxA(f \rightarrow g)$ *dans une fonction* `f` *est la liste des couples* $(Ch_{p_i}, \mathcal{P}_{eff_i})$ *de chacun des contextes d'appel de la fonction imbriquée. Soit la fonction* `g` *à* n *contextes d'appel différents dans la fonction* `f`*, l'ensemble des contextes d'appel de la fonction imbriquée se définit comme :*
>
> $$CtxA(f \rightarrow g) = [(Ch_{p_1}, \mathcal{P}_{eff_1}), \ldots, (Ch_{p_n}, \mathcal{P}_{eff_n})]$$

Notation 20
L'extraction du i^{eme} élément d'une liste L se note $elem(L, i)$ ainsi l'extraction du i^{eme} contexte d'appel de la fonction imbriquée `g` dans la fonction `f` se note $elem(CtxA(f \rightarrow g), i)$ où $i \leq |CtxA(f \rightarrow g)|$.

```
1   int f(int x1, int x2)
2   {
3     int y;
4     if (x1>=0)
5     {
6       x1=x1+3;
7       x2=3*x2;
8       y=g(x1,x2);/*instruction d'appel */
9     }
10    else
11    {
12      x1=x1+7;
13      y=g(x2-x1,x2);/*instruction d'appel */
14    }
15    y=3*y;
16    return y ;
17  }
```

FIG. 9.3 – Fonction sous test avec plusieurs instructions d'appel

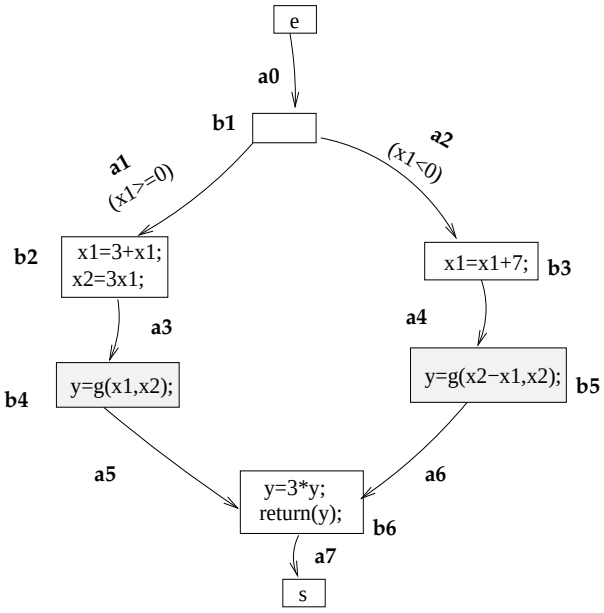

FIG. 9.4 – CFG de la fonction `f` de la figure 9.3

Illustration 64
Étudions la fonction `f` dont le code source est donné dans la figure 9.3, fonction contenant un appel imbriqué de la fonction `g`. Nous construisons dans un premier temps le graphe de contrôle de la fonction `f` dont le résultat est présenté dans la figure 9.4.

Il existe deux blocs d'appel différents de la fonction `g` *représentés par les blocs de base grisés dans la figure 9.4 (cf. les blocs* ***b4, b5*** *respectivement aux lignes 8 et 13 du code source associé). Pour chaque bloc d'appel, il existe un unique chemin dans la fonction* `f` *précédant l'appel soit un seul contexte d'appel :*

- *pour le bloc d'appel* ***b4*** *:*
  $(Ch_{p1}, \mathcal{P}_{eff_1})$ *où* $Ch_{p_1} = (e, a_0, b_1, a_1, b_2, a_3)$ *et* $\mathcal{P}_{eff_1} = [x1, x2]$ *et*
- *pour le bloc d'appel* ***b5*** *:*
  $(Ch_{p2}, \mathcal{P}_{eff_2})$ *où* $Ch_{p_2} = (e, a_0, b_1, a_2, b_3, a_4)$ *et* $\mathcal{P}_{eff_2} = [x2 - x1, x2]$.

On a donc

$$CtxA(f \rightarrow g) = [(Ch_{p_1}, \mathcal{P}_{eff_1}), (Ch_{p_2}, \mathcal{P}_{eff_2})]$$

Nous venons de caractériser les contextes d'appel d'une fonction imbriquée dans le cas simple. Nous allons généraliser au cas où une structure répétitive précède une instruction d'appel.

9.2.2 Contextes d'appel et structure répétitive

Dans le cas où une structure répétitive précède l'exécution d'une instruction d'appel, chaque chemin d'exécution partiel possédant un nombre d'itérations différent dans la structure répétitive introduit un nouveau contexte d'appel pour la fonction imbriquée.

Une instruction d'appel d'une fonction imbriquée peut avoir un nombre élevé de chemins partiels la précédant dans la fonction appelante. Cependant, le nombre de chemins partiels faisables peut être potentiellement grand mais est toujours borné y compris en présence de structure répétitive précédant l'instruction d'appel. Nous insistons sur ce point car cela signifie que la liste contenant les contextes d'appel d'une fonction ne peut pas être une liste infinie. En effet, une infinité de chemins partiels précédant une instruction d'appel reviendrait à la présence d'une structure répétitive infinie avant cette instruction d'appel. Or, une structure répétitive infinie, par définition, ne se termine pas et par conséquent toutes les instructions succédant cette structure d'appel ne sont pas exécutées (y compris, pour le cas qui nous intéresse, l'instruction d'appel). De plus, comme nous l'avons expliqué, chaque chemin partiel précédant une instruction d'appel définit une partition sur le domaine d'entrée de la fonction imbriquée, l'ensemble de ces partitions étant exclusives. Une infinité de chemins partiels correspondrait à la caractérisation d'une infinité de partitions sur le domaine d'entrée de la fonction imbriquée qui est un domaine fini. De plus, rappelons également que l'application du critère des k-chemins borne à k le nombre de passages par structure répétitive.

```
1   int ff(int z1,int z2)
2   {
3    while (z1<0)
4        z1= z2+z1;
5    z1= 2*z2;
6    z2= gg(z1*z2,z2-z1);
7    return(z2);
8   }
```

FIG. 9.5 – Code source de la fonction sous test `ff`

La fonction `ff` dont l'implantation est donnée dans la figure 9.5 contient une structure répétitive précédant l'appel de la fonction imbriquée `gg`. Le graphe de flot de contrôle de la fonction `ff` associé se trouve dans la figure 9.6.

Nous allons définir $CtxA(ff \rightarrow gg)$ l'ensemble des contextes d'appel de la fonction `gg` dans la fonction `ff`. Nous identifions donc les chemins dans `ff` amenant à l'appel de `gg` :

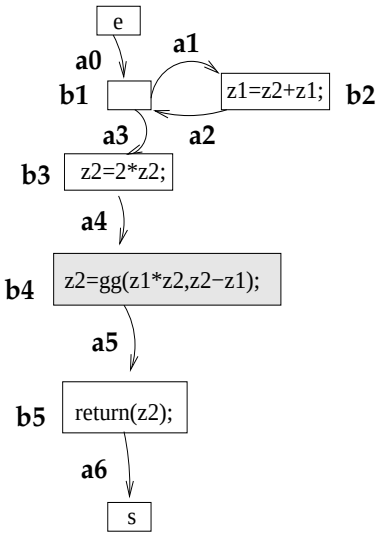

FIG. 9.6 – Graphe de contrôle de la fonction `ff`

- $Ch_{p_{ff_1}} = (e, a0, b1, a3, b3, a4, b4, a5, s)$ (pas d'itération dans la structure répétitive)
- $Ch_{p_{ff_2}} = (e, a0, b1, a1, b2, a2, b1, a3, b3, a4, b4, a5, s)$ (1 itération dans la structure répétitive)
- $Ch_{p_{ff_3}} = (e, a0, b1, a1, b2, a2, b1, a1, b2, a2, b1, a3, b3, a4, b4, a5, s)$ (2 itérations dans la structure répétitive)
- ...

La forme générale des chemins dans la fonction `ff` amenant à l'appel de `gg` est de la forme suivante : $Ch^*_{p_{ff}} = (e, a0, b1, (a1, b2, a2, b1)^*, a3, b3, a4, b4, a5, s)$. L'expression des paramètres effectifs de l'appel de `gg` est $\mathcal{P}_{eff} = [z1 * z2; z2 - z1]$ quelque soit le nombre d'itérations dans la structure répétitive précédant l'appel.

Nous obtenons donc :

$$CtxA(ff \rightarrow gg)^* = [(Ch^*_{p_{ff}}, \mathcal{P}_{eff})]$$

avec $[(Ch^*_{p_{ff}}, \mathcal{P}_{eff})]$ représentant par abus de notation la liste des couples $[(Ch^0_{p_{ff}}, \mathcal{P}_{eff}), (Ch^1_{p_{ff}}, \mathcal{P}_{eff}), \ldots, (Ch^k_{p_{ff}}, \mathcal{P}_{eff})]$. Chaque chemin partiel de la fonction sous test `ff` ayant un nombre d'itérations différent dans la structure répétitive précédant l'appel introduit un nouveau contexte d'appel pour la fonction imbriquée `gg`.

Quand un chemin partiel de la fonction sous test amenant à un bloc d'appel contient lui-même une autre instruction d'appel, l'analyse se complique. Dans un tel cas, d'un point de vue purement structurel, chaque chemin de la fonction imbriquée faisable contenu dans un des chemins partiels de la fonction appelante introduit un nouveau contexte d'appel de la fonction imbriquée. En pratique, pour une méthode de test "inlining", chaque bloc d'appel est remplacé par l'expression des chemins structurels imbriqués précédée de l'affectation de ses paramètres formels et succédés par l'affectation des variables de sortie de la fonction imbriquée [2].

Nous ne nous attardons par sur ce point car l'abstraction des chemins internes aux fonctions imbriquées que nous proposons dans le chapitre 11 écarte cette difficulté.

9.3 Domaine d'appel d'une fonction imbriquée

Chaque contexte d'appel d'une fonction imbriquée caractérise une partition de son domaine en entrée sur laquelle la fonction imbriquée va être exécutée. Cela nous amène à définir cette

[2] Nous avons expliqué ce point dans la section 2.6.5 du chapitre 2.

partition du domaine d'entrée comme le domaine d'appel $DomA(g, elem(CtxA(f \rightarrow g), i))$ pour $i \leq |CtxA(f \rightarrow g)|$ d'une fonction imbriquée g pour un contexte d'appel donné.

Convention 8
Pour alléger la lecture et les notations, nous désignerons par la suite le domaine d'appel d'un i^{eme} contexte d'appel $DomA(g, elem(CtxA(f \rightarrow g), i))$ par la notation simplifiée suivante

$$DomA_i(g, f \rightarrow g)$$

pour tout i entier tel que $1 \leq i \leq |CtxA(f \rightarrow g)|$.

Remarque(s) 27
Nous rappelons que la notation $D1_{|D2}$ représente la restriction du domaine $D1$ au domaine $D2$.

Illustration 65
Nous avons déjà illustré la notion de domaine d'appel d'une fonction sans en avoir utilisé la terminologie exacte. L'illustration 63 à la page 140 décrit la caractérisation du domaine d'appel de la fonction `g` dans la fonction `f` pour un contexte d'appel donné $CtxA(f \rightarrow g) = [(Ch_p, \mathcal{P}_{eff})]$. On obtient pour cette illustration :

$$DomA(g, f \rightarrow g) = [16, MaxInt] \times ([MinInt, MaxInt - 16]) \times [35, MaxInt]$$

Les valeurs en entrée de la fonction imbriquée sont issues des valeurs des variables de la fonction appelante ou de variables globales. L'exécution du chemin partiel dans la fonction appelante restreint le domaine de ces variables au codomaine de ce chemin partiel. Le domaine d'appel correspond au domaine en entrée de la fonction pour des variables en entrée dont les domaines sont réduits au codomaine du chemin partiel.

Le domaine d'appel d'une fonction $DomA_i(g, f \rightarrow g)$ est un sous-domaine de $Dom(g)$, le domaine de la fonction g.

Le domaine d'appel d'une fonction g dans une fonction `f` pour un contexte d'appel donné $elem(CtxA(f \rightarrow g), i) = (Ch_p, \mathcal{P}_{eff})$ pour $i \leq |CtxA(f \rightarrow g)|$ correspond à :

$$DomA_i(g, f \rightarrow g) = Dom(g)_{|CoDom(Ch_p)}{}^{3}$$

Notons que le fait de couvrir tous les chemins de la fonction appelante revient à exécuter tous les chemins partiels précédant une instruction d'appel. Par conséquent, toutes les fonctions imbriquées sont appelées sur chacun de leurs contextes d'appel et donc sur chacun de leurs domaines d'appel.

9.4 Graphe d'appel

Notre représentation de graphe d'appel[4] modélise, comme tout graphe d'appel (nous verrons cette notion plus précisément dans la section 10.1.2) différentes relations entre la fonction sous test et les fonctions imbriquées. Pour notre représentation, une relation considérée entre entre deux fonctions est un contexte d'appel reliant ces deux mêmes fonctions.

Soit l'ensemble F dans lequel sont énumérées les fonctions imbriquées dans la fonction sous test. Si nous reprenons le code de la fonction sous test `f` dont l'implantation est donnée dans la figure 9.3, nous avons donc $F = \{g\}$. Pour chaque fonction imbriquée, il s'agit ensuite de déterminer tous les contextes d'appel.

A partir de ces informations, nous pouvons construire le **graphe d'appel** de la fonction appelante.

[3] Sous réserve que le codomaine de Ch_p est convenablement projeté sur les bonnes variables d'entrée de la fonction `g`.

[4] Notre notion de graphe d'appel diffère de la notion usuelle : il s'agit en réalité d'un arbre et non d'un graphe mais nous avons choisi de réutiliser la terminologie habituelle.

Définition – 9.4.1

*Un **graphe d'appel** d'une fonction `f` est le graphe connexe orienté* $GAppelP :< f, F, A, \delta >$ *où :*

- F *est l'ensemble composé des fonctions imbriquées dans la fonction* `f`,
- A *est une relation binaire de* f *vers* F *représentant les différents contextes d'appels entre la fonction sous test et les fonctions imbriquées et*
- δ *est la fonction d'étiquetage* $\delta : (\delta_F, \delta_A)$ *où*

$$\delta_F : F \rightarrow L_F$$

$$\delta_A : A \rightarrow L_A$$

où L_F *(resp.* L_A*) l'ensemble des fonctions imbriquées associées à l'ensemble* F *(resp. l'ensemble des contextes d'appel de la fonction* `f` *vers les fonctions imbriquées associé à* A*).*

La présence d'un arc $(Ff1, Ff2)$ dans un graphe d'appel signifie que la fonction `f2` associée au nœud $Ff2$ est imbriquée dans la fonction `f1` associée au nœud $Ff1$ c'est-à-dire que la fonction `f1` contient une instruction d'appel pour `f2`. L'étiquetage de cet arc, $\delta_A(Ff1, Ff2)$ contient le contexte d'appel associé.
Les figures 9.7 et 9.8 contiennent respectivement les graphes d'appel des fonctions `f` et `ff` dont l'implantation est donnée dans les figures 9.3 et 9.5 et dont les contextes d'appel des fonctions imbriquées ont été définis respectivement dans l'illustration 64 de la page 141 et dans la section 9.2.2.

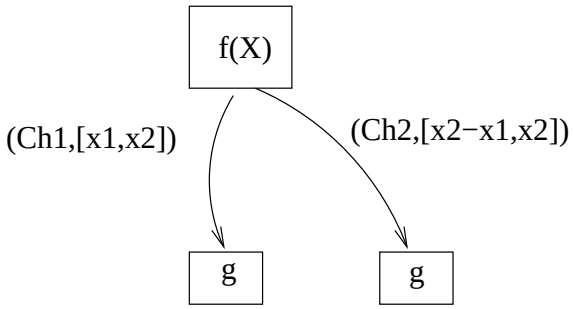

FIG. 9.7 – Graphe d'appel de la fonction `f`

Le graphe d'appel de la fonction `ff` faisant intervenir une structure répétitive avant l'instruction d'appel et dont le code se trouve dans la figure 9.5 se trouve dans la figure 9.8.

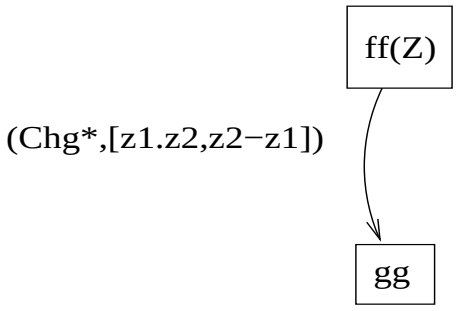

FIG. 9.8 – Graphe d'appel de la fonction `ff`

9.5 Principe de base

Nous nous plaçons dans une stratégie ascendante de test c'est-à-dire que les fonctions imbriquées ont déjà été testées unitairement et jugées conformes à leurs spécifications. Notre idée est d'utiliser les spécifications exprimées sous forme de couples pre/post (cf. chapitre 3) des fonctions imbriquées afin de les abstraire pour limiter leur exploration tout en maintenant la couverture structurelle de la fonction appelante.

Notre stratégie de gestion des appels de fonction proposée dans ce manuscrit s'inscrit donc dans le couplage des techniques de test fonctionnel et structurel.

En effet, nous proposons d'appliquer une méthode de test mixte destinée au traitement des fonctions imbriquées et d'incorporer cette gestion des appels à la méthode PathCrawler présentée dans le chapitre 7.

Nous avons choisi de conserver et d'exécuter les fonctions imbriquées dans la fonction sous test afin de conserver le comportement réel de la fonction sous test et de ne perdre aucune information. Nous abstrayons les fonctions imbriquées par leurs spécifications lors du parcours des chemins exécutés et du calcul des prédicats de chemins associés dans la fonction sous test.

Notre objectif est d'assurer la couverture complète des chemins faisables de la fonction sous test (c'est-à-dire la fonction appelante) tout en limitant l'exploration des fonctions imbriquées pour éviter l'explosion combinatoire des chemins.

Nous détaillerons de façon précise notre approche dans le chapitre 11. Avant cela, dans le chapitre suivant, nous allons dresser un panorama des techniques existantes pour le traitement des fonctions imbriquées et nous nous situerons par rapport à ces différentes techniques.

Chapitre 10

État de l'art et positionnement

Nous allons dans ce chapitre dresser un rapide panorama des techniques existantes ayant pour objectif de tester une entité (fonction, composant) s'insérant dans un ensemble d'entités (programme, système à base de composants).

10.1 Généralités

Commençons tout d'abord par rappeler quelques notions vues dans le chapitre 5.

10.1.1 Test unitaire, test d'intégration et test imbriqué

Le test unitaire a pour objectif de vérifier que chaque composant individuel[1] fonctionne correctement, indépendamment des autres composants et ce, sur l'ensemble de son domaine de définition.

Le test d'intégration consiste, quant à lui, à tester l'agencement des différents composants d'un système ainsi que la façon dont ils communiquent entre eux [Bei90]. Son principal objectif est de détecter les problèmes d'interface entre les modules composant un logiciel afin de mettre en évidence des défauts de communication ou d'interdépendance (comme les accès aux données, la transmission des données, la détection d'effets de bords ou encore, selon la nature du logiciel, la synchronisation, etc) de modules validés individuellement avant leur intégration.

Introduisons maintenant un cas particulier de technique de test à savoir le **test imbriqué (ou en contexte)**.

Le test imbriqué (ou test en contexte) a pour but de vérifier qu'un composant possède un comportement à l'exécution conforme à celui de sa spécification lorsqu'il interagit avec d'autres composants. Cette catégorie de test s'adresse aux systèmes reposant sur de multiples interactions et communications entre modules (comme les protocoles de communication, les systèmes de contrôles avioniques,...).

Ainsi, le test d'intégration s'oriente plus vers le test d'un groupement de composants testés unitairement alors que le test imbriqué s'oriente vers le test unitaire d'un composant donné dans un environnement précis c'est-à-dire au sein du groupement de composants auquel il appartient.

[1] la plus petite entité de code compilable et exécutable, ce qui, pour notre approche, désigne les fonctions (avec retour) et/ou procédures (sans retour) du langage C

10.1.2 Graphe d'appel

Lors du test d'intégration, une des premières étapes est de construire le graphe d'appel de l'ensemble des composants du programme sous test.

Il existe de multiples façons de représenter un graphe d'appel de fonction. Il s'agit dans un premier temps d'identifier clairement ce que l'on entend par relation entre composants. En effet, selon le type de relations entre composants auquel on s'intéresse pour le test d'intégration, une relation pourra désigner :

- un appel simple entre deux composants c'est-à-dire pour le langage C, toute instruction d'appel (cf. section 2.6.3) d'une procédure sans variable de sortie comme par exemple l'appel d'une procédure d'écriture dans un fichier,
- un appel avec utilisation des sorties (le composant appelé est soit une fonction, soit une procédure dont les variables de sortie sont réutilisées par le composant appelant),
- un partage de variables (deux fonctions utilisant et/ou définissant une même variable globale sans instruction d'appel de l'une des fonctions à l'autre),
- une communication indirecte (fonctions accédant à un même fichier),
- etc.

Une fois la notion de relation entre fonctions clairement définie, le **graphe d'appel** consistera en un graphe connexe orienté dont les nœuds seront les différents composants du programme sous test et les arcs les relations entre ces composants clairement définies au préalable.

Convention 9
Comme nous l'avons vu dans la définition 9.4.1 de notre graphe d'appel, nous nous intéressons aux relations entre composants définies par chaque contexte d'appel des fonctions imbriquées dans la fonction sous test.

La représentation courante est de relier les fonctions dont l'une possède dans son corps une instruction d'appel à la seconde. Ainsi un arc (Nf, Ng) indique que la fonction `f` associée au nœud Nf fait appel à la fonction `g` associée au nœud Ng.

Définition – 10.1.1

> *Une **graphe d'appel** simple d'une fonction `f` est le graphe connexe orienté* $G =< Nf, N, E >$ *avec :*
>
> - *Nf le nœud d'entrée de G associé à la fonction `f`,*
> - *N l'ensemble des nœuds du graphe représentant les fonctions imbriquées utilisées par la fonction `f` et*
> - *E, relation binaire de Nf vers N, l'ensemble des arcs du graphe représentant les différentes instructions d'appel.*

Remarque(s) 28
Par la suite, sans autre précision de notre part, la notion de graphe d'appel désignera la définition 9.4.1 plus riche que la définition 10.1.1.

Illustration 66
Le graphe d'appel de la fonction `fA` de la figure 10.1, avec `fA` la fonction associée au nœud d'entrée A, indique que les fonctions associées aux nœuds B et C sont appelées par la fonction `fA` et que la fonction associée au nœud B appelle également la fonction associée au nœud D.

Les feuilles d'un graphe d'appel représentent les fonctions de plus bas niveau dans la hiérarchie du logiciel. La racine du graphe ou nœud d'entrée représente la fonction racine du logiciel c'est-à-dire la fonction de plus haut niveau dans la hiérarchie du logiciel.

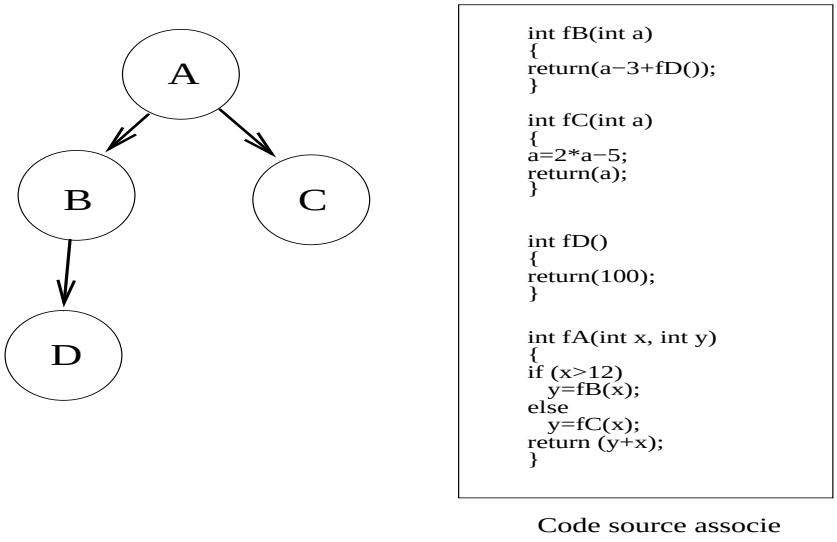

FIG. 10.1 – Graphe d'appel simple de la fonction `fA`

10.1.3 Différents types de bouchons

Lors du test d'intégration, les fonction testées ne peuvent pas être exécutées en isolation. Il est ainsi souvent indispensable de pouvoir simuler les fonctions imbriquées encore non intégrées et/ou de stimuler les entrées et sorties de la fonction sous test. L'intégration progressive des fonctions requiert donc l'utilisation et le développement de composants logiciels conçus pour être substitués lors de l'assemblage aux fonctions encore non développées ou intégrées. Ces composants logiciels supplémentaires sont de deux types :

– **les bouchons** («stubs») pour remplacer les fonctions imbriquées encore non intégrées et
– **les lanceurs** («drivers») pour remplacer les fonctions appelantes encore non intégrées.

Pour les bouchons, il faut distinguer les bouchons dits structurels des bouchons dits fonctionnels.

Un **bouchon structurel** est un composant logiciel se substituant à la fonction imbriquée associée qui retourne toutes les valeurs en sortie forçant l'exécution de l'ensemble des chemins structurels de la fonction sous test y compris ceux pouvant être en réalité infaisables (le comportement de la fonction imbriquée est donc ignoré ici).

Illustration 67
Avec le critère des branchements appliqué à la section de code suivante :

```
if (f(x) <0 )
  /*bthen*/
else
  /*belse*/
```

il faut couvrir au moins une fois la branche accédant au bloc de base bthen et la branche accédant au bloc de base belse.

L'expression $(f(x) < 0)$ *est remplacée par le bouchon structurel* $(bf(x))$ *qui retournera* 0 *puis* 1 *afin d'atteindre la couverture donnée indépendamment du vrai comportement de la fonction remplacée.*

L'utilisation de bouchons structurels requiert la construction d'un bouchon pour chaque bloc d'appel de la fonction imbriquée (y compris les blocs d'appel succédant une structure répétitive)

sauf si la fonction qu'il remplace est passive i.e. qu'il s'agit d'une procédure ne définissant aucune variable.

Les propriétés de la fonction remplacée ne sont pas utilisées lors de la création du bouchon associé, seule l'objectif de la couverture de la fonction appelante est pris en compte. Le problème de ce type de bouchon est qu'il peut amener à couvrir des chemins infaisables en réalité.

Illustration 68
Reprenons l'exemple précédent de l'illustration 67, la fonction `f` *est en réalité la fonction de calcul du carré d'une valeur telle que* $f(x) = x^2$. *L'expression* `(f(x)<0)` *n'est jamais vérifiée alors que le bouchon structurel va forcer la couverture des deux branches de la conditionnelle y compris la branche infaisable.*

Un **bouchon fonctionnel** est un composant logiciel exécutable qui simule le vrai comportement de la fonction qu'il remplace en fournissant les mêmes sorties pour les mêmes entrées.

Le bouchon fonctionnel est plus complexe à mettre en œuvre que le bouchon structurel mais plus rigoureux car il s'agit d'un composant logiciel ayant la même interface et les mêmes **réactions** vis-à-vis des appels extérieurs qu'une fonction donnée. Son but est de remplacer ce dernier en tant que composant dans l'assemblage : le bouchon étant jugé comme fiable car construit à partir de spécifications validées, toute erreur ne peut lui être attribuée et l'intégration composant par composant permet la localisation rapide des erreurs dans l'assemblage.

10.1.4 Lanceurs

Un **lanceur** est un composant logiciel ayant la même interface et les mêmes **actions** d'appel qu'une fonction donnée. Il a pour objectif de remplacer une fonction appelante dans l'assemblage.

Illustration 69
Prenons le code suivant :

```
int f(int x)
{
  if (x<10)
      x=g(x);
  else
     return(x);
}
```

et que nous intégrons la fonction `g` *avant la fonction* `f` *alors cette dernière fonction est remplacée par un lanceur qui fera appel à la fonction* `g`.

10.1.5 Graphe d'accessibilité

Pour le test imbriqué c'est-à-dire le test d'un composant donné dans un système à base de composants, le système est modélisé sous forme d'un automate composé d'un nombre fini d'états. Chaque composant représente un état donné du système. Deux états du système sont généralement distingués à savoir l'état initial dans lequel le système est dit "en attente" et l'état final dans lequel le système est arrêté (ne peut plus prendre un nouvel état) ou dans lequel le système redevient "en attente" (il s'agit du cas le plus général où l'état initial et l'état final sont confondus). L'automate modélisant le système sous test correspond à un graphe dont les nœuds représentent les états du systèmes et les arcs représentent les actions à effectuer pour passer d'un état à l'autre. Cet automate est appelé graphe d'accessibilité et correspond à un système de transitions étiqueté dont tous les états sont accessibles par un chemin de longueur finie depuis au moins le nœud d'entrée correspondant à l'état initial du système.

Définition – 10.1.2

*Un **graphe d'accessibilité** $G =< Ni, Nf, N, E, \delta >$ est un graphe connexe orienté étiqueté avec :*

- *N un ensemble fini de nœuds associés aux différents états du programme,*
- *E un ensemble de N vers N représentant les transitions (actions) permettant le passage d'un état à un autre état,*
- *δ la fonction d'étiquetage $\delta : (\delta_N, \delta_E)$ avec*

$$\delta_N : N \rightarrow L_N$$

$$\delta_E : E \rightarrow L_E$$

avec L_N (resp. L_E) l'ensemble des états associés à l'ensemble N (resp. l'ensemble des actions provoquant des transitions d'états associées à E),

- *le nœud d'entrée Ni associé à l'état initial du système avec $Ni \in N$*
- *le nœud de sortie Nf associé à l'état final du système (souvent confondu avec l'état initial).*

Illustration 70

S'il existe un arc $(n_i, n_{i+1}) \in E$ et n_i et $n_{i+1} \in N$ alors si le système se trouve dans l'état associé à n_i et que l'action associée à l'arc (n_i, n_{i+1}) est exécutée alors la transition est activée et le système passe dans l'état n_{i+1}.

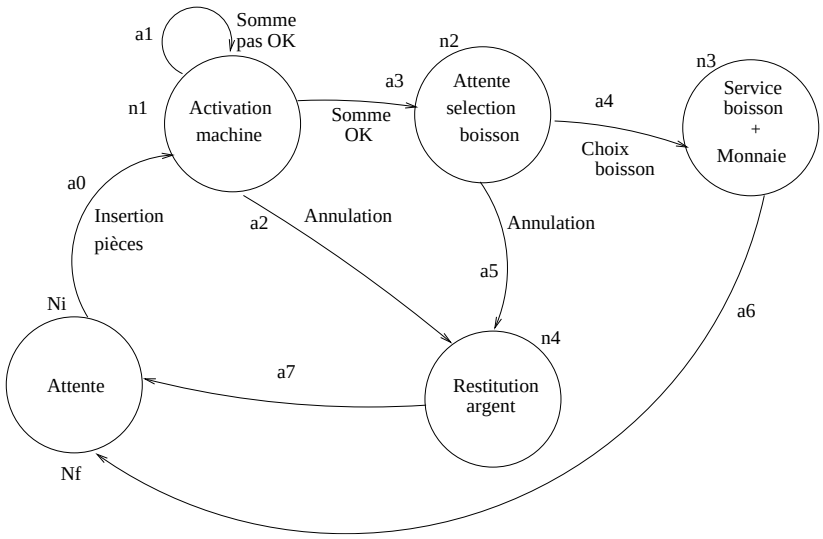

Fig. 10.2 – Graphe d'accessibilité simplifié pour un système modélisant une machine à café

La figure 10.2 contient le graphe d'accessibilité d'un système modélisant le fonctionnement d'une machine à café.

Intuitivement, le graphe d'accessibilité, en général cyclique, est le graphe minimal (en nombre de transitions et d'états) représentant tous les comportements possibles (i.e. toutes les suites de transitions) d'un système. Pour un nombre fini d'états n (5 pour l'exemple associé à la figure 10.2) correspond un nombre n de nœuds dans le graphe d'accessibilité et pour chaque transition entre deux états (8 pour l'exemple associé à la figure 10.2) correspond un arc dans le graphe. Les arcs redondants sont ensuite éliminés.

Pour des systèmes complexes, les graphes d'accessibilité associés sont de grande taille. La difficulté de tester un composant donné du système correspond à l'atteignabilité de ce composant

sous test. A partir de l'état initial du système, il s'agit de déterminer la séquence des transitions permettant d'atteindre le composant voulu, d'activer l'état associé à ce composant et de remettre le système dans un état stable à savoir dans l'état initial.

10.2 Test imbriqué ou en contexte

Les techniques de test imbriqué s'adressent en particulier aux systèmes contenant des composants fortement connectés et dont la communication entre composants joue un rôle prépondérant. Ces techniques se justifient sur deux aspects principaux :

- la complexité du système sous test ne permet pas de le tester dans sa totalité,
- l'importance des interactions entre composants ne permet pas de les tester unitairement et indépendamment les uns des autres.

Le test imbriqué consiste donc à tester un composant enfoui dans un environnement appelé contexte (incluant le composant sous test) jugé correct.

10.2.1 La mise en place de techniques de test imbriqué

Un des problèmes majeurs de ces approches concerne la construction du graphe d'accessibilité du fait que l'explosion combinatoire résultant de la construction et en particulier de l'exploration du graphe d'accessibilité limite l'applicabilité de la méthode comme l'explique [YCA99]. Le problème de la construction du graphe d'accessibilité touche la plupart des techniques de test imbriqué. Selon un critère donné (définissant les objets à couvrir), il s'agit alors de déterminer les chemins et les séquences de tests correspondantes dans le graphe d'accessibilité.

Définition – 10.2.1

*Une **séquence de test** atteignant un état associé à n_i est un chemin du nœud initial Ni au nœud final Nf du graphe d'accessibilité $G =< Ni, Nf, N, E, \delta >$ contenant le nœud n_i correspondant à la séquence suivante :*
$(n_0, (n_0, n1), n1, \ldots, (n_{i-1}, n_i), n_i, (n_i, n_{i+1}), \ldots, (n_{n-1}, n_n), n_n)$
où $n_0 = Ni$, $n_n = Nf$ et $n_j \in N$ où $(n_j, n_{j+1}) \in N$ pour $j \in [0, .., n-1]$ et $n_j \in N$.
Une séquence de test se décompose en trois parties :

- *le sous-chemin de Ni à la transition précédant l'état sous test n_i correspondant à la séquence $(n_0, (n_0, n1), n1, \ldots, (n_{i-1}, n_i))$ se nomme le préambule de la séquence,*
- *le corps de test qui est le nœud associé à l'état sous test n_i et*
- *le postambule est le sous-chemin de la transition succédant l'état sous test à l'état final correspondant à la séquence $((n_i, n_{i+1}), \ldots, (n_{n-1}, n_n), n_n)$.*

Illustration 71

On reprend le système représenté par la figure 10.2. On veut tester l'état «Restitution argent» associé au nœud $n4$.
Un préambule correspond à atteindre un état précédent l'état à tester (c'est-à-dire un nœud prédécesseur du nœud de l'état à tester) comme le nœud $n2$ correspondant à l'état «Attente sélection boisson» par exemple $(Ni, a0, n1, a3, n2)$.

Le corps de test consistera à activer la transition $a5$ «Annulation» qui nous amène à l'état voulu $n4$ soit $(a5, n4)$.

Le postambule consiste à revenir à l'état final «Attente» et la transition $a7$ est automatique donc $(a7, Nf)$.
Une séquence de test est donc $(Ni, a0, n1, a3, n2, a5, n4, a7, Nf)$.

Une séquence de test correspond à une suite d'actions à générer pour parcourir un chemin donné du graphe d'accessibilité et ensuite d'identifier les entrées correspondantes à chaque action.

Les entrées induisant cette séquence d'actions sont obtenues comme pour toute génération de cas de test par une analyse statique du code du système ou par une analyse de ses spécifications.

Convention 10
Par la suite, nous entendrons par séquence de test, la séquence des valeurs en entrée entraînant les actions activant les transitions atteignant l'état cible.

10.2.2 Construction des séquences de test

Les méthodes de test imbriqué s'appuient sur différentes techniques. [YCA99] et [LSKP96] utilisent, entre autres techniques de génération de cas de test, **la minimisation des séquences de test** (par l'identification des transition redondantes ou inutiles entre autres), **la génération des séquences de test à la volée** comme dans [FJJV96] qui fonctionne selon un parcours en profondeur d'abord du graphe avec la prise en compte des transitions déjà ou non parcourues et la recherche du plus court chemin par la construction de la séquence de test.

Les algorithmes les plus couramment utilisés se distinguent pour leur stratégie de sélection des séquences de test. Ils correspondent à l'application d'un critère structurel sur le graphe d'accessibilité. On parle d'**algorithme structurel**. Dans cette section, tous les travaux sont des algorithmes structurels est le critère choisi est «toutes les transitions» du graphe d'accessibilité (équivalent au critère toutes-les-branches si on projette sur un graphe de contrôle). Il s'agit alors à partir du nœud initial de trouver le chemin (ou l'ensemble des chemins) le plus court permettant de couvrir au moins une fois toutes les transitions. Même si la construction du graphe ne pose pas de problème dans certains cas, la recherche du plus court chemin reste un problème NP-difficile.

Une première technique est la **marche aléatoire** [MC94] qui, comme son nom l'indique, consiste à parcourir le graphe d'accessibilité aléatoirement. Il s'agit d'une technique à la volée : la construction totale du graphe est évitée. Cependant les séquences de test générées ne sont pas optimisées. De plus, cette technique impose de manipuler des séquences de test souvent conséquentes du fait qu'elles soient aléatoires et qu'il faille atteindre un état donné du système.

Ces deux algorithmes sont en pratique négligés au profit d'algorithmes plus efficaces dont nous allons donner quelques exemples.

La marche aléatoire guidée

Une variante est la **marche aléatoire guidée** [LSKP96] qui fonctionne de la même façon mais avec la possibilité d'orienter l'exploration du graphe selon les transitions déjà parcourues. Les séquences de test sont évidemment plus courtes que dans la technique totalement aléatoire mais restent non optimisées puisque cela reste un problème NP-complet. Le choix de la transition est là encore aléatoire mais s'effectue parmi l'ensemble des transitions restant à tester dans un premier temps et si cet ensemble est insuffisant pour l'état courant, une transition déjà parcourue est activée.

Algorithme "Hit-or-Jump"

Enfin, une dernière technique nommée **"Hit-or-Jump"** [CLRZ99] s'apparente un peu à un couplage des techniques de la marche aléatoire et de la marche aléatoire guidée. Il s'agit d'une recherche locale autour d'un état courant dans le graphe réduit à une taille de voisinage défini par l'utilisateur. Si ce voisinage comporte des transitions non testées, on parcourt celles-ci (on parle de "Hit" qui correspond à une technique orientée) sinon on écarte du test tout ce voisinage (on parle de "Jump" qui correspond à un déplacement aléatoire dans le graphe). Malheureusement, il s'agit d'une heuristique dont la méthode peut échouer mais celle-ci permet tout de même d'éviter la construction totale du graphe contrairement à l'algorithme structurel et permet, dans le cas idéal, d'obtenir des séquences de test plus courtes que la technique de marche aléatoire. La recherche locale permet de diminuer les séquences ("Hit") mais celles-ci sont amplifiées par les phénomènes aléatoires ("Jump").

10.2.3 Mise en parallèle avec nos besoins

Notre objectif est de limiter la combinatoire des chemins dans les fonctions imbriquées et par conséquent le nombre de cas de test tout en maintenant la couverture de tous les chemins faisables de la fonction sous test. Essayons de faire un parallèle entre le test imbriqué et notre objectif.

Si on assimile notre graphe d'appel de la fonction sous test et un graphe d'accessibilité, les transitions représentent les différentes instructions d'appel. Une couverture de tous les arcs (transitions) du graphe d'accessibilité est équivalente à une couverture de tous les arcs du graphe d'appel d'une fonction c'est-à-dire une couverture de **toutes les instructions d'appel de toutes les fonctions imbriquées** du graphe d'appel si nous raisonnons en termes de graphe.

Cela correspond donc à couvrir toutes les instructions d'appel de la fonction sous test au moins une fois. Or, selon les conditions d'appel d'une fonction imbriquée, plusieurs chemins en sortie de la fonction imbriquée peuvent être exercés : la simple exécution de chaque instruction d'appel est donc insuffisante pour garantir le maintien de la couverture de la fonction appelante. Cette technique ne permet pas de garantir la couverture de 100% des k-chemins faisables de la fonction sous test. L'application de cette stratégie ne s'adapte donc pas à nos besoins.

10.3 Test d'intégration

C'est lors de la phase de test d'intégration que seront prises en compte les fonctions préexistantes implantées lors d'un précédent projet et réutilisées pour le logiciel sous test. Une erreur alors souvent observée est une utilisation de la fonction hors domaine c'est-à-dire un appel de fonction avec des valeurs en entrée n'appartenant pas au domaine de définition de la fonction.

Les fonctions imbriquées sont ainsi testées dans leurs réels contextes d'utilisation y compris l'utilisation possible hors domaine alors que le test unitaire d'une fonction se limite à la couverture de son domaine de définition.

Peu de méthodes sur le test d'intégration sont formalisées. En effet, on observe généralement une extension peu appropriée des méthodes de test unitaire au test d'intégration.

10.3.1 Granularité considérée

On parle de test d'intégration pour de nombreux types de test. Tout dépend de la granularité considérée. En effet, le test d'intégration a pour objectif de tester simultanément différents composants communiquant entre eux. On peut considérer un composant comme étant une fonction, une procédure donnée d'un logiciel ou une classe, la méthode d'une classe pour les langages orientés objets [JE94]. On peut également discerner différents niveaux de test d'intégration comme le test d'intégration composant/composant, le test d'intégration logiciel/composant ou encore le test d'intégration logiciel/logiciel.

Pour notre stratégie, nous nous situons dans l'optique composant/composant avec la notion de composant désignant une fonction et/ou une procédure du langage C comme définies dans le chapitre 2.

Rappelons que le test d'intégration ne consiste pas à répéter les tests unitaires sur des groupes de composants mais de tester leur composition. On remarque cependant de nombreuses techniques de test d'intégration dérivant des techniques de test unitaires.

10.3.2 Méthodes dérivées des méthodes unitaires fonctionnelles

La plupart des techniques de test d'intégration sont des techniques reposant sur l'étude des spécifications des différents modules. Elles s'apparentent beaucoup aux techniques unitaires de test fonctionnelles.

Les jeux de tests sont définis par l'étude des documents de conception et/ou de spécification. Ils ont pour objectif de vérifier la bonne intégration des composants intégrés à la dernière étape.

Le principal avantage d'une méthode fonctionnelle est de pouvoir choisir le degré de précision selon le degré de précision de la décomposition fonctionnelle. L'inconvénient majeur du test d'intégration fonctionnel est de ne tester que les comportements spécifiés et non tous les comportements réels d'une fonction. Il est en effet possible qu'une erreur d'implantation introduise un nouveau comportement de la fonction différent de celui spécifié ou absent de la spécification. Une technique fonctionnelle peut ainsi ne pas détecter un comportement réel de la fonction si celui-ci est non spécifié.

Méthodes dites "category-partition"

Les méthodes **"category-partition"** sont les techniques de test d'intégration parmi les plus courantes. Elles reposent sur un découpage fonctionnel de la fonction selon ses différentes fonctionnalités [OB88]. Il s'agit d'identifier l'ensemble des variables d'entrées influençant ces fonctionnalités et de générer les différentes cas de test en faisant varier méthodologiquement les valeurs de l'ensemble de ses variables d'entrée. Les différentes fonctionnalités du programme permettent de définir les catégories de la fonction qui, elles-mêmes, seront partitionnées en différentes classes d'équivalences sur les entrées ou **"choices"**. Pour chaque partition, les classes d'équivalences sont disjointes mais leur union doit couvrir totalement le domaine des variables d'entrée de la partition.

Une **catégorie d'une fonction** est un sous-domaine d'une fonction choisi selon un critère précis (le domaine d'une des variables d'entrée de la fonction sous test par exemple)

Une **classe d'équivalence d'une fonction** est un sous-domaine d'une catégorie de fonction contenant des valeurs des variables d'entrée ayant des propriétés communes et un même traitement dans la fonction.

Les étapes (toutes manuelles) de cette méthode sont au nombre de cinq :
- analyse des spécifications afin d'identifier les différentes unités (composants) du logiciel,
- identification du domaine d'entrée en terme de paramètres et de variables d'environnement qui modifient chaque unité fonctionnelle,
- identification des catégories correspondant à un partitionnement de la fonction correspondant à une fonctionnalité donnée selon les variables en entrée par exemple,
- partitionnement des catégories précédemment définis en classes d'équivalences ou "choices" et
- spécification des combinaisons possibles des "choices" à tester.

Seul le dernier point est modifié selon le critère de test choisi.

Si nous prenons comme composant une fonction F, une catégorie correspond au domaine de chacune de ses variables d'entrée et chaque catégorie est partitionnable en classes d'équivalence pour chaque sous-domaine des variables d'entrée correspondant à un des domaines fonctionnels de la fonction. Le système est supposé se comporter de manière similaire pour tous les éléments d'une même classe. De plus, si une catégorie possède l classes d'équivalences alors l'union de ces l classes d'équivalence couvre exactement le domaine de la catégorie associée et chaque classe d'équivalence définit un sous-domaine de la catégorie.

De plus, les classes d'équivalence sont disjointes entre elles.

Ainsi, pour une fonction `f` de profil

$$f : s_1 \dots s_n \rightarrow s'_1$$

nous pouvons définir n catégories (une par variable d'entrée) et définir différents sous-domaines de ces catégorie (un par domaine fonctionnel de la fonction). Chaque sous-domaine correspondant à une classe d'équivalence de la fonction.

Illustration 72
Exemple extrait de [OB88] :

Une fonction de tri prend en entrée une variable représentant un tableau d'éléments de taille variable et de type indéfini et retourne la permutation des éléments du tableau de façon à ce qu'ils soient triés selon un critère prédéfini (cette fonction ne possède donc qu'un seul domaine fonctionnel). La fonction produit également deux sorties parallèles retournant les éléments minimum et maximum du tableau. Les catégories identifiées sont alors la taille du tableau, le type des éléments, la valeur maximale et minimale ainsi que la position initiale de ces deux derniers éléments. Une possibilité de partitionner la catégorie «taille du tableau» peut être :

- *taille=0,*
- *taille=1,*
- *2≤taille≤100 et*
- *taille>100*

en se basant sur les observations d'erreurs typiques des tableaux. Si on prend en compte que la mémoire allouée pour des tableaux de taille variable correspond à des blocs de 256, on peut aussi partitionner la catégorie «taille du tableau» selon que sa taille soit inférieure, égale ou supérieure à 256.

L'avantage de cette approche est de reposer sur une étude rigoureuse de l'influence des variables d'entrée. On peut critiquer le fait que la recherche d'erreurs ne soit pas orientée vers la communication et les connexions entre modules c'est-à-dire vers les points que le test d'intégration doit justement éprouver.

10.3.3 Méthodes dérivées des méthodes unitaires structurelles

Une autre optique est de dériver des techniques de test unitaire en construisant les cas de test sur une étude du code source des composants sous test. Ces techniques, requérant plus de cas de tests, interviennent souvent pour les logiciels dont le niveau de confiance exigé est grand.

Les techniques reposent généralement sur le principe de l'étude du flot de contrôle ou du flot de données [Spi92] afin de tester un maximum de séquences d'appels de fonctions et de mettre en évidence les anomalies des flots de contrôle ou de données apparaissant aux interconnexions des modules. Pour cela, il s'agit de construire, par exemple, le graphe de flot de contrôle de la fonction appelante totalement déplié c'est-à-dire en remplaçant les appels de fonctions par le graphe de flot de contrôle des fonctions imbriquées. Dans ce graphe déplié, on supprime tout chemin ne passant par aucune transition d'appel comme le montre la figure 10.3. Ce sous-graphe ainsi extrait peut alors être soumis à critère structurel habituel.

Un critère de test choisi peut être, par exemple, «DD-paths» (chemins de décision à décision, pour des décisions consécutives) le test de tous les chemins possibles entre deux instructions conditionnelles du code source.

Illustration 73
Si on regarde la figure 10.3 (le graphe de contrôle déplié de la fonction imbriquée est repérable par ses nœuds sur fond grisé), il existe 3 nœuds décisionnels : 1,2 et 1' après suppressions des chemins ne passant par aucune transition d'appel. L'application du critère «DD-paths» consistera donc à couvrir tous les chemins reliant ces différents nœuds entre eux i.e. tous les chemins contenant l'arc $(1,2)$ *ou les séquences* $((2,3),3,(3,1'))$, $((2,4),4,(4,1'))$.

Une variante est le critère «MM-paths» [Jor85] (chemins de module à module) correspondant à la couverture de tous les arcs entrants et sortants des nœuds imbriqués lors du dépliage de la fonction appelante.

Illustration 74
Toujours en se basant sur la figure 10.3, il s'agit de couvrir tous les chemins entre deux modules ce qui signifie tous les chemins contenant les arcs reliant deux modules entre eux, à savoir tous les chemins contenant $(3,1')$, $(4,1')$, $(5,1')$ *et* $(3',6)$. *Les nœuds décisionnels sont les nœuds* $1,2,1'$.

Ce critère correspond dans notre cas à couvrir toutes les instructions d'appels et retour de fonction. Nous ne voulons pas nous contenter de couvrir tous les appels de fonctions et tous les

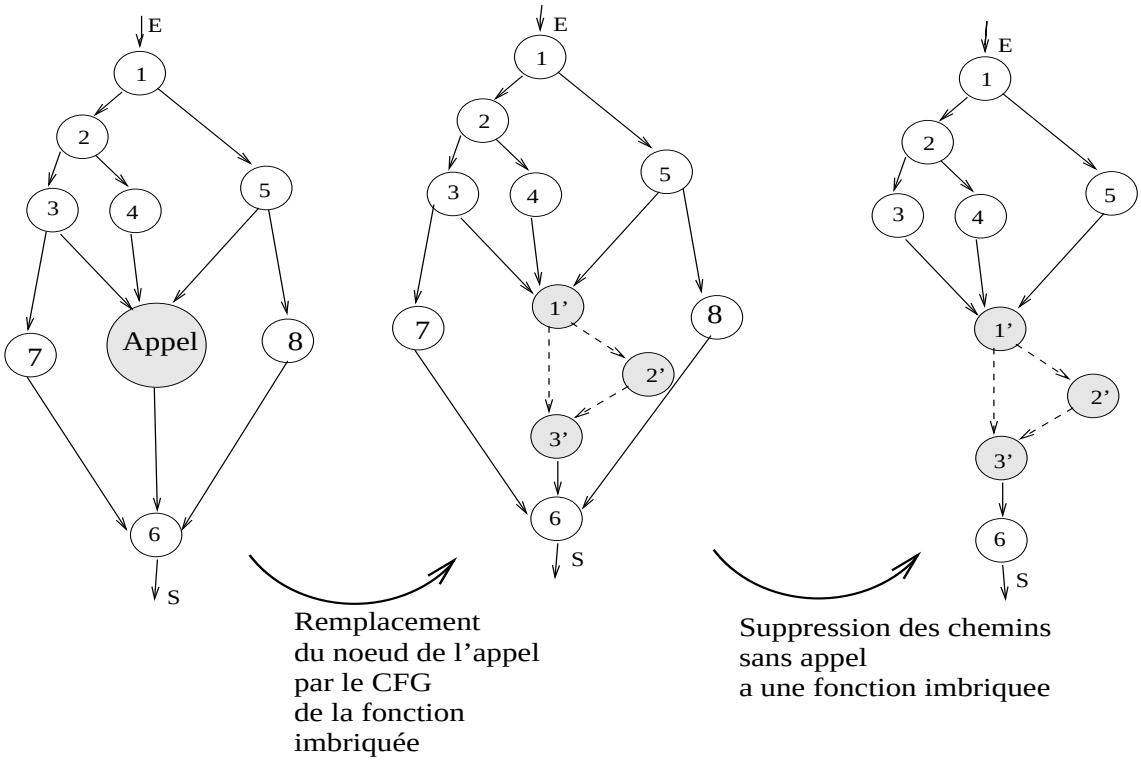

FIG. 10.3 – Exemple de graphe de contrôle avec dépliage de la fonction imbriquée.

retours mais tous les chemins faisables ce qui correspond à couvrir toutes les **combinaisons possibles des appels et retours des fonctions imbriquées** pour les chemins contenant des instructions d'appels tout en limitant la combinatoire des chemins des fonctions imbriquées.

Selon le type de logiciel testé, on pourra aussi choisir de couvrir tous les accès aux données concurrents ou tous les cas de synchronisation, etc.

Une autre stratégie que l'on peut citer comme dérivant du test unitaire est le test des valeurs aux limites des domaines [LW90] que ce soit pour les méthodes structurelles ou fonctionnelles. En effet, il s'avère qu'une grande proportion des erreurs se trouve aux bords des domaines : une erreur typique lors de l'implantation est par exemple d'utiliser une inégalité stricte au lieu d'une inégalité simple ou inversement.

Cette technique d'une part élimine des chemins structurels de la fonction appelante (ce que nous ne voulons pas) et s'apparente beaucoup à un traitement " inlining" des fonctions imbriquées pouvant entraîner rapidement une explosion combinatoire des chemins et par conséquent du nombre des cas de test à effectuer.

10.3.4 Méthodes dites "coupling-based"

Les méthodes que nous venons de voir dans la section précédente dérivent de méthodes de test unitaire structurel et peuvent être remises en question dans la mesure où les différents cas de test ne sont pas déterminés par une étude en vue de trouver les erreurs d'interconnexions des différents modules d'un programme ou d'un système.

Il existe cependant un type de méthodes structurelles d'intégration n'étant pas dérivées du test unitaire mais orientées vers les connexions entre modules d'un programme.

Tout d'abord, le code source du programme est analysé afin d'identifier les différentes relations entre les divers composants. La méthode repose donc sur une technique structurelle basée sur une classification en niveaux pour la communication entre les fonctions. Ces niveaux sont définis en fonction :

- des appels de composants,
- du partage de variables globales,
- de passage de variables en paramètres, etc.

Offutt, Harold et Kolte vont même jusqu'à une distinction entre les usages des paramètres entre les modules (utilisation lors d'un calcul, lors d'une décision ou de façon indirecte) [OHK93] :

- l'utilisation de la valeur d'une variable dans un calcul comme dans une instruction, dans l'expression d'une sortie ou dans l'appel d'un autre composant,
- l'utilisation d'une variable dans une condition [FW88],
- l'utilisation de la valeur d'une variable pour la modification de la valeur d'une seconde variable utilisée par la suite dans une condition [OHK93].

Ils proposent une classification en 12 niveaux différents dépendant du type d'utilisation des variables entre les modules, comme par exemple :

- "independant coupling" : deux modules indépendants (aucune relation les liant) et
- "call-coupling" : un des composants appelle le second sans partager aucune variable ni paramètre.
- ...

Par souci de simplification, ces 12 niveaux peuvent cependant être regroupés en quatre sous-groupes. Ainsi, deux modules seront dits liés différemment s'ils partagent ou non une variable, s'ils communiquent via un objet externe comme un fichier. Ces relations entres les modules permettent de construire un graphe de relation dont les nœuds représentent un ensemble fini de fonctions et les arcs un ensemble des relations entre deux fonctions. Pour cette représentation il faut construire un graphe pour chaque type de relation prise en compte entre les fonctions.

Il s'agit alors de définir les objets du graphe à couvrir par le choix d'un critère de test (tous les arcs, les nœuds, ...). Cette méthode est justifiée dans la mesure où le but est de tester les interconnexions des composants et que les différents cas de test dépendent directement du type de ces interconnexions.

Dans [JO95], une confrontation des résultats des méthodes de type "category-partition" et "coupling-based" sur une même fonction donne la seconde méthode comme plus efficace (meilleure détection des erreurs pour moins de cas de test et moins de temps de génération).

Le test d'intégration se faisant en ajoutant des modules à un ensemble déjà testé, le type d'assemblage des modules (un par un ou par groupes de modules) ainsi que l'ordre de leur intégration permettent de différencier les différentes méthodes existantes.

10.3.5 Intégration par incrément ou par agrégats

Lors d'un test d'intégration, il faut faire un choix entre intégrer un module à la fois ou intégrer progressivement entre eux des groupes de composants. On parle respectivement d'**intégration par incrément** et d'**intégration par agrégats.**

L'intégration par incrément possède l'avantage d'une meilleure localisation des erreurs. En effet, de façon générale, l'erreur sera due au dernier composant intégré. En revanche, une telle démarche demande un nombre de tests plus élevé car les étapes intermédiaires seront plus nombreuses. Il faudra aussi considérer la création importante de logiciels bouchons pour simuler les modules pas encore intégrés.

Notons qu'il est possible d'intégrer en une seule fois tous les composants mais cela rend particulièrement difficile la localisation des différentes erreurs, on parle alors d'**intégration massive** ou "big-bang".

10.3.6 Méthodes d'intégration ascendantes et descendantes

Une fois tous les modules identifiés et le graphe d'appel construit, il faut choisir un ordre d'assemblage. Il s'agit d'un choix délicat qui doit prendre en compte la disponibilité des composants, la nécessité ou non de création de logiciels bouchons et/ou lanceurs ainsi que l'architecture du logiciel testé.

La méthode **ascendante ou "bottom-up"** repose sur une agrégation en priorité des modules les plus bas dans la hiérarchie d'appels, l'intégration se fait à partir des feuilles du graphe d'appel. Les modules font tous, au préalable, l'objet de tests unitaires[2]. Nous considérons le test unitaire des modules comme la première étape du test d'intégration. Ensuite les différents modules vont être intégrés par étapes intermédiaires en remontant dans le graphe d'appel. Son avantage est d'éliminer le nombre de bouchons dans la mesure où les fonctions de bas niveau sont intégrées avant les fonctions de plus haut niveau. Cette démarche nécessite cependant la création de nombreux logiciels lanceurs afin de simuler l'appel des modules intégrés. Le coût de conception d'un composant lanceur est moindre par rapport à celui d'un composant bouchon mais nécessite tout de même un travail d'analyse non négligeable : il remplace la fonction appelante donc et doit donc appelé la fonction sous test avec les bons arguments.

Convention 11
Dans les trois exemples qui vont suivre nous allons nous appuyer sur le figure 10.1 représentant le graphe d'appel de la fonction `fA` *représenté par le nœud A et utilisant les fonctions* `fB, fC, fD` *représentées respectivement par les nœuds B, C, D. Pour simplifier les exemples, nous confondrons les nœuds avec les fonctions associées ce qui signifie que A désignera à la fois le nœud A et la fonction* `fA`.

Illustration 75
En reprenant la figure 10.1, la méthode "bottom-up" consiste en l'ordre d'intégration suivant : D, C, B, A. Il faut créer, dans l'ordre, un lanceur pour D, un lanceur pour C puis un lanceur pour B et C.

La méthode **descendante ou "top-down"** débute quant à elle par les modules de plus haut niveau c'est-à-dire à partir de la racine du graphe d'appel. La détection des problèmes d'architecture est plus précoce que la méthode précédente et les jeux de test peuvent être réutilisés pendant les différentes étapes intermédiaires. En effet, les composants de plus hauts niveaux sont tout d'abord intégrés et ce sont ces composants qui prennent en entrée les jeux de test pour appeler les autres composants. Les jeux de test de la première phase du test d'intégration peuvent donc être réutilisés aux phases suivantes afin de mettre à l'épreuve les derniers composants intégrés et donc de plus bas niveau. Cette démarche demandera la création de nombreux bouchons (au moins un par bloc d'appel dans la fonction sous test).

Illustration 76
Toujours avec la figure 10.1, l'ordre d'intégration de la méthode descendante est inversée par rapport à la méthode ascendante. Il faut donc créer cette fois un bouchon pour B, un bouchon pour C et un bouchon pour D.

Une autre optique possible est de coupler les deux précédentes méthodes, on parle alors d'**intégration en sandwich**. L'ordre d'intégration dépend alors du contexte. On peut par exemple choisir d'intégrer au plus tôt les composants jugés les plus critiques (susceptibles de posséder plus d'erreurs) afin d'identifier au plus vite les erreurs et limiter ainsi le coût de la correction des problèmes. L'idée est, dans ce cas, d'affronter au plus tôt les difficultés attendues mais demande également la création de nombreux logiciels bouchons et de logiciels lanceurs. Ici,

[2] Lors du test unitaire de fonctions du programme, des lanceurs et bouchons sont créés pour chacune des fonctions du programme.

la logique de l'architecture n'est pas prise en compte pour les différents cas de test.

Illustration 77
En reprenant la figure 10.1 et en considérant que l'ordre de criticité des modules est : C, B, D, A. L'ordre d'intégration est C, B, D, A , il faut donc créer un lanceur pour C, un lanceur pour B et C , un bouchon pour D puis un lanceur pour B et C et un lanceur pour D.

Remarque(s) 29
Plus les modules intégrés sont complexes, plus la création de bouchons est coûteuse. Il vaut mieux donc privilégier la création de lanceurs et donc une méthode "bottom-up" ou alors pour des modules critiques clairement identifiés une méthode en sandwich basée sur l'ordre de criticité des modules.

Toutes ces méthodes peuvent avoir une justification selon les objectifs du test, la criticité et la complexité des modules. Il s'agit donc d'étudier la plus pertinente pour un contexte donné.

10.4 Une méthode originale de gestion des appels

Ce manque d'outillage quant à la gestion des appels de fonction empêche le passage à l'échelle pour des fonctions réalistes. [God07] s'intéresse au problème du passage à l'échelle des méthodes structurelles de test unitaires dynamiques de test en présence d'appels de fonction. L'idée est de tester unitairement les fonctions de bas niveau (les fonctions imbriquées) selon le principe présenté dans [GKS05] et de conserver toutes les informations issues de ce test unitaire des fonctions imbriquées en termes des contraintes sur les valeurs en entrée et en sortie pour chaque chemin des fonctions imbriquées. Ces informations constituent un "résumé" du fonctionnement des fonctions imbriquées. Ainsi, lors du test d'une fonction utilisant une de ces fonctions, les fonctions imbriquées sont abstraites par la disjonction des contraintes sur les valeurs en entrée et en sortie de la fonction pour chacun des chemins couverts précédemment. Notons que [God07] ressemble à notre approche à la différence près qu'elle utilise des couples "pre/post structurels" et que nous utilisons des couples pre/post fonctionnels d'abstraction supérieure et complets sur le domaine de définition de la fonction imbriquée. De plus, le test unitaire de [GKS05] se limite à la couverture de chemins dont la longueur maximale a été fixée, cela signifie les chemins faisables non couverts (car de longueur trop grande) ne sont pas pris en compte dans le "résumé" des fonctions imbriquées.

10.5 Parallèle avec les techniques existantes pour la gestion des appels de fonction

10.5.1 Maintien de la couverture de la fonction sous test

Utilisation de bouchons structurels

Une optique est de remplacer les appels de fonctions par des bouchons structurels. Une technique qui, comme nous l'avons expliqué un peu avant dans la section 10.1.3, possède le défaut de se concentrer uniquement sur la fonction sous test et sur le critère de test appliqué sans prendre en compte le comportement réel des fonctions imbriquées. De plus, cette technique nécessite l'implantation d'un bouchon structurel pour chacun des contextes d'appel des fonctions imbriquées et peut amener à couvrir des chemins infaisables.

Incorporation des sorties des appels de fonction comme entrées de la fonction sous test

Une autre technique possédant les mêmes inconvénients est de définir **les sorties des fonctions imbriquées comme des variables d'entrée de la fonction sous test**. Cela permettrait d'obtenir 100% des chemins faisables de la fonction sous test sans explosion combinatoire. Les sorties des fonctions imbriquées seront là aussi calculées uniquement en vue d'atteindre une couverture donnée de la fonction sous test et non de refléter le comportement réel des fonctions imbriquées. Le traitement des sorties des fonctions imbriquées comme des entrées de la fonction sous test introduirait de plus un grand nombre de chemins supplémentaires inexistants dans la fonction sous test ou chemins infaisables en réalité. Comme pour les bouchons structurels, le problème majeur est que les sorties de la fonction sous test seront non conformes à la réalité. Cela nous empêche alors un verdict de test car on ne peut savoir si le comportement de la fonction sous test est non valide à cause d'un défaut de la fonction sous test ou au comportement non conforme d'un bouchon par rapport à la fonction qu'il remplace.

Pour toutes ces raisons, nous avons exclu ces deux techniques de notre analyse.

10.5.2 Prise en compte du comportement des fonctions imbriquées

Une autre approche est de procéder au test unitaire de la fonction en utilisant des **bouchons fonctionnels** pour les appels de composants (cf. section 10.1.3).

Bouchon fonctionnel et combinatoire des chemins

Nous réutilisons la fonction `f` modélisée dans la figure 9.1 de la page 138 telle que $f(X) = Y$ et possédant un bloc d'appel pour la fonction imbriquée g où n chemins exécutables partiels dans la fonction `f` amènent à l'appel de `g` et m chemins exécutables succèdent l'appel de la fonction imbriquée `g` jusqu'à la sortie de la fonction `f`. En utilisant un bouchon fonctionnel basé sur les domaines fonctionnels de la fonction imbriquée `g`, si nous supposons que cette fonction possède x domaines fonctionnels cela signifie que $n*m*x$ chemins structuro-fonctionnels sont alors à couvrir.

Une spécification se situe à un niveau d'abstraction supérieur au code implanté. Le test unitaire avec bouchons fonctionnels pour les appels de composant permet de limiter la combinatoire des chemins si nous émettons l'hypothèse suivante :
à un domaine fonctionnel de la spécification d'une fonction correspond au moins un chemin structurel dans le graphe de contrôle de la fonction.
Cette hypothèse est crédible dans la mesure où chacune des fonctionnalités attendues du programme doit être implantée dans le code source et que deux fonctionnalités distinctes dans une spécification ne peuvent pas, en pratique, correspondre à un même chemin d'exécution dans le graphe.

Bouchon fonctionnel vs. traitement "inlining"

Selon l'hypothèse précédente, l'utilisation de bouchons fonctionnels limite la combinatoire des chemins. En effet, pour p le nombre de chemins structurels de la fonction imbriquée et x le nombre de ses domaines fonctionnels où $x \leq p$, le nombre de chemins dans la fonction sous test à couvrir peut être multiplié par p avec un traitement "inlining" et par x par l'utilisation d'un bouchon fonctionnel.

Dans le meilleur des cas, la combinatoire est limitée par l'utilisation d'un bouchon fonctionnel par rapport à un traitement "inlining" et dans le pire des cas, la combinatoire n'est pas amplifiée par rapport au dépliage de la fonction c'est-à-dire si à chaque domaine fonctionnel correspond exactement un chemin structurel dans le graphe de la fonction imbriquée (toujours selon la même hypothèse raisonnable en pratique).

Bouchon fonctionnel et maintien de la couverture de la fonction sous test

L'abstraction des chemins internes aux fonctions imbriquées permet le limiter l'explosion combinatoire des chemins par rapport à un traitement "inlining". Cependant, cette technique possède le désavantage de ne pas garantir, du fait de l'abstraction, le maintien de la couverture de la fonction sous test. En effet, les chemins structurels de la fonction sous test sont abstraits par la séquence d'un chemin partiel structurel précédant l'appel, un domaine fonctionnel de la fonction imbriquée et un chemin partiel structurel succédant cet appel. Or, il est possible que pour un chemin partiel structurel précédant un appel peuvent être activés plusieurs domaines fonctionnels de la fonction imbriquée et de même, il est possible que pour un domaine fonctionnel activé puissent être exécutés plusieurs chemins structurels en sortie. Il faut alors pouvoir orienter l'exécution des bouchons fonctionnels pour pouvoir garantir le maintien de la couverture de la fonction sous test. Or, les bouchons fonctionnels peuvent être vus comme des modules "boites noires" retournant pour une entrée donnée, la sortie attendue conforme à la spécification ainsi que la relation entrées/sorties exercée : la relation entrées/sorties est ainsi vérifiée par construction. L'utilisation de bouchons fonctionnels ne prend pas en compte la possibilité de plusieurs chemins exécutables en sortie de la fonction imbriquée qu'il remplace ce qui peut amener à ne pas couvrir certains chemins faisables de la fonction appelante.

10.5.3 Positionnement

Le but est de tester la fonction dans son contexte réel d'utilisation sans rajouter de chemins supplémentaires aux chemins exécutables de la fonction sous test. Il faut cependant pouvoir garantir le maintien de la couverture de 100% des chemins faisables de l'appelant pour atteindre notre objectif. Nous nous sommes donc basés pour notre stratégie sur les techniques "inlining" et basées sur l'utilisation de bouchons fonctionnels afin de tirer parti de leurs avantages respectifs et de répondre à leurs inconvénients.

Notre approche peut s'apparenter à une méthode de test d'intégration dans laquelle les fonctions appelées sont substituées par des bouchons fonctionnels. Nous verrons dans le chapitre suivant que nous allons être amenés à couvrir chaque domaine fonctionnel des fonctions imbriquées pour chacun de leurs contextes d'appel. Le fait que les fonctions imbriquées soient exécutées dans leur contexte réel (l'abstraction des fonctions imbriquées n'intervient qu'au moment de la modélisation des chemins couverts) apparente également cette méthode à une méthode de test imbriqué.

Maintenant que nous avons positionné notre approche quant à la gestion des appels de fonction par rapport aux techniques existantes de test d'intégration et de test imbriqué, nous allons pouvoir décrire plus en détails l'approche que nous proposons dans le chapitre suivant.

Chapitre 11

Description et mise en œuvre

Nous allons décrire dans ce chapitre la modélisation que nous proposons pour les chemins structurels contenant des instructions d'appel. Nous allons nous attarder sur chaque étape ainsi que sur la mise en œuvre de notre stratégie. Nous rappelons que notre objectif est de maintenir la couverture des k-chemins de la fonction sous test tout en limitant au maximum l'exploration des fonctions imbriquées.

11.1 Prétraitement des fonctions sous test avec appels imbriqués

Notre méthode s'adresse aux fonctions sous test implantées en langage C contenant des appels de fonction. Cependant, il est important de rappeler que le code source testé subit au préalable un prétraitement simplifiant son analyse.

Parallèlement à la mise en place des instructions de trace, le code source de la fonction sous test est soumis au prétraitement expliqué dans la section 7.2. Le point sur lequel nous allons nous attarder concerne le traitement des expressions à effets de bord dont l'instruction d'appel fait partie. Comme expliqué dans la section 7.2.6, après le prétraitement des codes sources effectué par CIL [Lan06], chaque instruction possède au plus une expression à effets de bord.

```
1  int somme(int a, int b)
2  {
3    int y;
4    y=a+b;
5    return(y);
6  }
7  int f(int x,int y)
8  {
9    y=somme(x,y);        /*simple appel*/
10   if (somme(x,y)>100) /*condition de branchement sur un appel*/
11     x=2*somme(y,x+y); /*produit d'un appel de fonction*/
12   else
13     y=2+somme(x,x+y); /*somme d'un appel de fonction*/
14 }
```

FIG. 11.1 – Fonction C avec instructions d'appel

Soit le fichier implanté en langage C contenant une fonction imbriquée `somme` et une fonction appelante `f` de la figure 11.1. Ce fichier contient différentes instructions d'appel de fonction à savoir un simple appel, une condition de branchement et différentes opérations sur le retour de la

fonction imbriquée. La figure 11.2 contient le résultat du prétraitement du code source du fichier de la figure 11.1.

Les appels de fonction sont extraits des conditions de branchement et chaque instruction d'appel est de la forme `tmp=somme(...);`.

Illustration 78
L'instruction `x=2*somme(y,x+y);` à la ligne 11 de la figure 11.1 est transformée en la séquence d'instructions (lignes 14 et 15 de la figure 11.2) :

```
tmp =somme(y, x + y);
x = 2 * tmp;
```

où la variable `tmp` est une variable créée lors du prétraitement.

Le retour des fonctions imbriquées est affecté à une variable de l'appelant potentiellement créée lors du prétraitement.

```
1  int somme(int a , int b )
2  { int y ;
3      y = a + b;
4    return (y);
5  }
6  int f(int x , int y )
7  { int tmp ;
8    int tmp___0 ;
9    int tmp___1 ;
10   y =somme(x, y); /*pas de modification notable*/
11   tmp___1 =somme(x, y);/*extraction de l'appel de fonction
12                                    de la condition de branchement */
13   if (tmp___1 > 100) {
14     tmp =somme(y, x + y); /*appel de fonction isolé*/
15     x = 2 * tmp;       /* produit sur le retour de la fonction */
16   }
17   else
18   {
19     tmp___0 =somme(x, x + y);/*appel de fonction isolé*/
20     y = 2 + tmp___0;       /* somme sur le retour de la fonction */
21   }
22   return (0);
23 }
```

FIG. 11.2 – Résultat du prétraitement sur le fichier de la figure 11.1

Remarque(s) 30
Par souci de lisibilité, nous avons ôté les instructions de trace rajoutées lors du prétraitement.

Un appel de fonction étant une expression à effets de bord, toute instruction d'appel ne peut être présente que dans une instruction séquentielle simple et par conséquent ne peut pas être utilisée dans une condition de branchement.

11.2 Graphe abstrait

Comme nous l'avons vu, le dépliage du graphe de contrôle des fonctions imbriquées dans le graphe de contrôle de la fonction sous test augmente de façon significative le nombre de chemins

de la fonction sous test. Le critère des k-chemins peut alors être rapidement inapplicable sur des fonctions réalistes. Nous désirons donc construire un graphe propre aux fonctions imbriquées d'un niveau d'abstraction supérieur par rapport à un graphe de contrôle et ce, en s'appuyant sur les spécifications des fonctions imbriquées.

11.2.1 Principe

Nous voulons conserver la notion et les informations des chemins structurels dans la fonction sous test tout en abstrayant les chemins structurels des fonctions imbriquées par des "chemins fonctionnels" issus de l'analyse de la spécification des fonctions imbriquées.

L'idée est d'abstraire les chemins structurels imbriqués par l'expression des couples pre/post associés dans la spécification de la fonction imbriquée. Nous construisons donc un graphe abstrait tel que, pour chaque couple pre/post dans la spécification d'une fonction imbriquée correspond un unique "chemin fonctionnel" dans notre graphe.

Ce graphe abstrait des fonctions imbriquées doit pouvoir se substituer au bloc d'appel des fonctions imbriquées dans le graphe de contrôle de la fonction sous test. Comme le graphe de contrôle d'une fonction, le graphe abstrait est un graphe connexe, orienté, étiqueté et possédant un unique nœud d'entrée et un unique nœud de sortie.

11.2.2 Caractérisation du graphe abstrait d'une fonction

De façon générale, comme pour un code source, nous allons représenter les spécifications d'une fonction imbriquée g sous forme de graphe. Nous partons d'une représentation proche du graphe de flot de contrôle d'une fonction à savoir un graphe connexe orienté avec une unique entrée et une unique sortie. Les conditions de branchements correspondent aux conditions sur les entrées de la fonction imbriquée des différents couples pre/post exprimées par les contraintes $Pre(g, W) \wedge D_i(g, W)$.

Remarque(s) 31
Nous raisonnons toujours avec l'hypothèse que $Q_i(g, W, Z)$ ne contient pas de contraintes supplémentaires sur W.

Reprenons les propriétés imposées aux spécifications explicitées dans le chapitre 3 à partir de la page 51.
La spécification d'une fonction étant complète selon la définition 3.5.1, au moins un couple pre/post est activé à l'appel de la fonction imbriquée : un ensemble de contraintes $Pre(g, W) \wedge D_i(g, W)$ est vérifié avec l'hypothèse de bonne utilisation de la fonction c'est-à-dire sans utilisation de la fonction imbriquée hors domaine. Il existe donc au moins un chemin exécutable dans le graphe abstrait d'une fonction imbriquée pour chacun de ses appels.

Les domaines fonctionnels de la spécification sont exclusifs entre eux afin d'éviter toute spécification contradictoire (cf. définition 3.5.2). Un seul et unique chemin du graphe abstrait peut être activé pour chaque appel d'une fonction imbriqué.

Les nœuds représentent des meta-instructions de la forme $FIND(Z|Q_i(g, W, Z))$ caractérisant la relation entrées/sorties définie par les contraintes contenues dans les $Q_i(g, W, Z)$. La spécification est déterministe selon la définition 3.5.3. Les meta-instructions $FIND(Z|Q_i(g, W, Z))$ correspondant à des prédicats existentiels de Z déterminent une unique instanciation de Z vérifiant $Q_i(g, W, Z)$ pour une instanciation donnée de W vérifiant $Pre(g, W) \wedge D_i(g, W)$.

Ainsi, à partir de chaque instanciation de W, il existe une unique instanciation de Z selon la sémantique de notre graphe abstrait.
Notre représentation des fonctions imbriquées sous forme de graphe abstrait correspond à un graphe de hauteur 2 avec n le nombre de couples pre/post dans la spécification comme le montre la figure 11.3. La sémantique de ce graphe correspond à la sémantique de la structure conditionnelle SwitchBreak du langage C donnée dans l'annexe A c'est-à-dire à une structure

conditionnelle à choix multiples exclusifs.

Définition – 11.2.1

Soit g une fonction définie par sa spécification $Spec(g, W, Z)$ à n couples pre/post. Le graphe abstrait de la fonction g associée à $Spec(g, W, Z)$ est le graphe étiqueté
$GAbs :< N, E, \delta, e, s >$ où e est l'unique nœud d'entrée et s l'unique nœud de sortie. N est un ensemble fini de $2 + n$ nœuds (dont les nœuds e et s). E est un ensemble de N vers N représentant les $2 * n$ arcs du graphe. La fonction d'étiquetage associée est $\delta : (\delta_N, \delta_E)$ telle que :

- $\delta_N : N \setminus \{e, s\} \rightarrow L_N$,
- $\delta_E : E_e \rightarrow L_E$ *où* E_e *représente les n arcs de la forme* (e, n_i) ,
- $\delta_E : E_s \rightarrow true$ *où* E_s *représente les n arcs de la forme* (n_i, s) *et* $E_e \cap E_s = E$.

avec L_N l'ensemble des meta-instructions $FIND(Z|Q_i(g, W, Z))$ et L_E l'ensemble des contraintes sur W $Pre(g, W) \wedge D_i(g, W)$ issues de $Spec(g, W, Z)$.

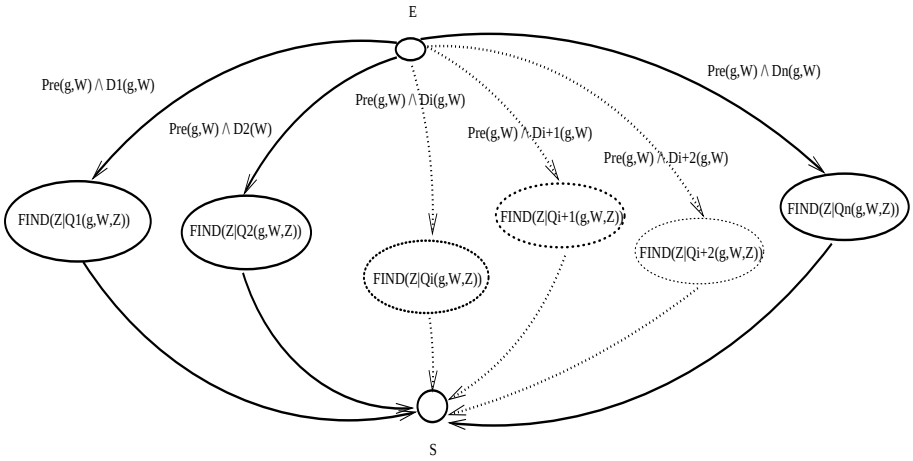

FIG. 11.3 – Représentation du graphe abstrait de la fonction g à partir de $Spec(g, W, Z)$

11.2.3 Deux Illustrations

La fonction valeur absolue `valAbs`

Nous nous appuyons sur la spécification de la fonction `valAbs` pour la construction de notre graphe abstrait :

$$Spec(valAbs, W, Z) = \{PP_1(valAbs, W, Z), PP_2(valAbs, W, Z)\}$$

$$PP_1(valAbs, W, Z) = (Pre(valAbs, W) \wedge D_1(valAbs, W), Q_1(valAbs, W, Z)) = (true \wedge w < 0, z = -w)$$

$$PP_2(valAbs, W, Z) = (Pre(valAbs, W) \wedge D_2(valAbs, W), Q_2(valAbs, W, Z)) = (true \wedge w \geq 0, z = w)$$

Les vecteurs W et Z tels que $p_1(W) = w$ et $p_1(Z) = z$ représentent respectivement les variables fonctionnelles en entrée et en sortie de la fonction imbriquée `valAbs`.

Selon les caractéristiques données dans la section précédente, nous pouvons construire le graphe abstrait de la fonction imbriquée *valAbs* tel qu'il est donné dans la figure 11.4.

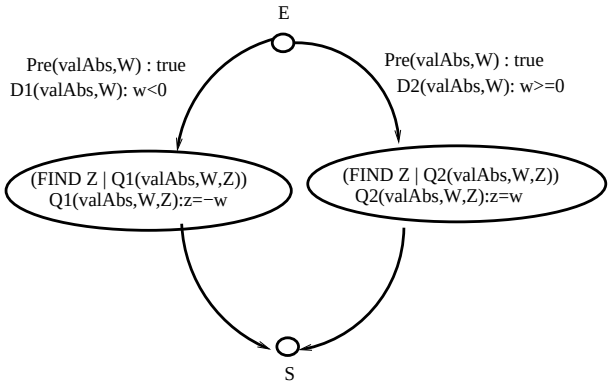

FIG. 11.4 – Graphe abstrait de la fonction `valAbs`

La fonction valeur médiane `getmid`

Nous reprenons l'exemple de la fonction `getmid`, utilisée dans la partie 2 de ce manuscrit comme une application à la méthode PathCrawler (cf. chapitre 7). Cette fonction retourne la valeur médiane des valeurs de ces trois variables d'entrée. Soient $p_1(W) = w1$, $p_2(W) = w2$, $p_3(W) = w3$ et $p_1(Z) = z1$, la spécification associée à cette fonction est la suivante :

$$Spec(getmid, W, Z) = \{PP_1(getmid, W, Z), PP_2(getmid, W, Z)\}$$

$$PP_1(getmid, W, Z) = (Pre(getmid, W) \wedge D_1(getmid, W), Q_1(getmid, W, Z))$$
$$PP_2(getmid, W, Z) = (Pre(getmid, W) \wedge D_2(getmid, W), Q_2(getmid, W, Z))$$
$$PP_3(getmid, W, Z) = (Pre(getmid, W) \wedge D_3(getmid, W), Q_3(getmid, W, Z))$$

---

où

$$Pre(getmid, W) = w1 \neq w2 \wedge w1 \neq w3 \wedge w2 \neq w3$$
$$D_1(getmid, W) = (w1 < w2 \wedge w2 < w3) \vee (w1 > w2 \wedge w2 > w3)$$
$$D_2(getmid, W) = (w2 < w1 \wedge w1 < w3) \vee (w2 > w1 \wedge w1 > w3)$$
$$D_3(getmid, W) = (w2 < w3 \wedge w3 < w1) \vee (w2 > w3 \wedge w3 > w1)$$
$$Q_1(getmid, W, Z) = (z1 = w2)$$
$$Q_2(getmid, W, Z) = (z1 = w1)$$
$$Q_3(getmid, W, Z)) = (z1 = w3)$$

Remarque(s) 32
Notons ici que nous excluons le cas où des variables en entrée de la fonction `getmid` ont la même valeur.

Le graphe abstrait associé à la fonction `getmid` est donné dans la figure 11.5.

Comme nous l'avons précisé au début de cette section, le graphe abstrait d'une fonction imbriquée a pour objectif de se substituer au graphe de contrôle de la fonction imbriquée lors du dépliage du CFG de la fonction sous test. Il s'agit donc de remplacer le bloc d'appel dans le CFG de la fonction sous test par le graphe abstrait de la fonction imbriquée. Il s'agit alors de faire les interconnexions entre les variables structurelles de l'appelante et les variables fonctionnelles de la fonction imbriquée.

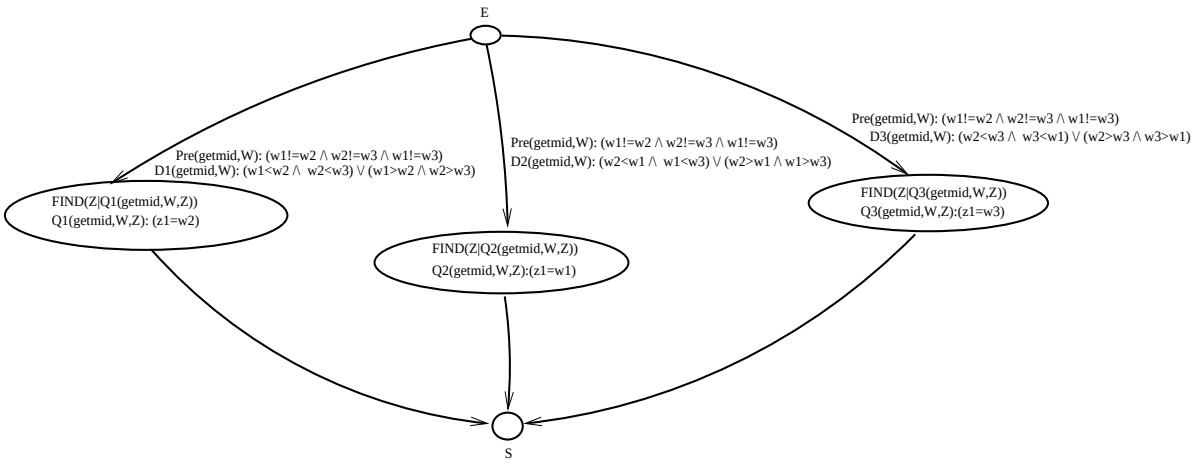

FIG. 11.5 – Graphe abstrait de la fonction `getmid`

11.3 Interconnexion du graphe de contrôle sous test aux graphes abstraits imbriqués

Nous allons expliquer dans cette section l'interconnexion du CFG de l'appelant aux graphes abstraits des fonctions imbriquées et plus précisément l'interconnexion entre les variables structurelles de l'appelant et les variables abstraites de la spécification des fonctions en question.

11.3.1 Introduction sur un exemple

Nous prenons un exemple faisant intervenir un appel de la fonction `valAbs`. Nous donnons tout d'abord le code source de la fonction appelante ainsi que celui de la fonction imbriquée dans la figure 11.6.

```
1  int f(int a, int b)
2  {
3  int x;
4  x=valAbs(a+b);
5  return(x);
6  }
7
8  int valAbs(int w)
9  {
10  int r;
11  if(w==0)
12      r=0
13  else
14    if (w<0)
15        r=-w;
16    else
17        r=w;
18  return r;
19 }
```

FIG. 11.6 – Fonction `f` sous test appelant la fonction `valAbs`

La figure 11.7 contient le graphe abstrait de la fonction `valAbs` ainsi que le graphe de contrôle

de la fonction appelante `f`.

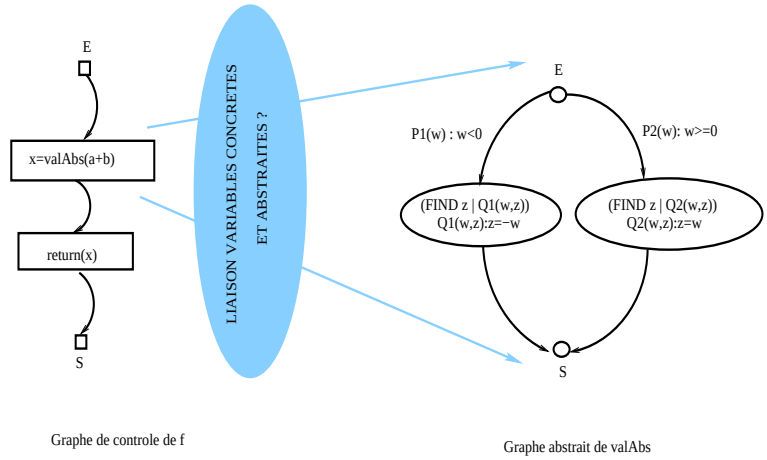

FIG. 11.7 – Graphe de contrôle de `f` et graphe abstrait de `valAbs`

On se rend compte ici que la difficulté consiste à fusionner les deux graphes en un afin d'obtenir notre graphe structuro-fonctionnel pour la fonction sous test et en particulier à mettre en correspondance les variables concrètes (dans le sens, issues de l'implantation) de `f` aux variables abstraites (dans le sens, issues de la spécification) de la fonction imbriquée.

Nous désignerons le graphe structuro-fonctionnel que nous allons construire comme le **graphe mixte** de la fonction sous test.

La modélisation de notre langage de spécification se veut être la plus proche possible de la modélisation du langage C. Ce point justifie la mise en correspondance des variables concrètes de la fonctions sous test aux variables abstraites des fonctions imbriquées.

De façon intuitive sur cet exemple, nous pouvons construire facilement le graphe mixte de la fonction sous test `f` en remplaçant l'instruction contenant l'appel de la fonction `valAbs` par :

- une instruction d'initialisation du paramètre d'entrée de `valAbs` représenté par la variable abstraite w par la valeur du paramètre effectif d'appel (pour l'exemple l'expression `a+b`),
- une instruction sous forme d'un Case[1] à choix multiples sur la valeur du paramètre d'entrée avec des conditions représentées par les $Pre(g, W) \wedge D_i(g, W)$ de $Spec(g, W, Z)$ et des meta-instructions basées sur les $Q_i(g, W, Z)$ permettant de déterminer la valeur de z et
- une instruction d'affectation de la valeur de sortie abstraite z à la variable `x` de `f`.

Convention 12
La sémantique du Case à choix multiples sur W correspond à tester W en le soumettant aux différentes conditions des couples pre/post de la spécification à savoir les $D_i(g, W)$ pour ensuite évaluer la meta-instruction associée de la forme :

$$FIND(Z|Q_i(g, W, Z))$$

ce qui permet avec les hypothèses sur $Spec(g, W, Z)$ de définir de façon unique les valeurs en sortie donc une instanciation de Z pour une instanciation donnée de W.

[1] Nous l'appelons ainsi car la sémantique du graphe abstrait est similaire de la sémantique de la structure SwitchBreak (appelée aussi Case).

Pour notre exemple, on obtient ainsi :

```
x=valAbs(a+b);          =>      w=a+b;
                                /* [ META-INSTRUCTIONS */
                                Case (P1(w):w<0):FIND(z|Q1(w,z):w=-z)
                               |Case (P2(w):w>=0) :FIND(z|Q2(w,z):w=z)
                                /* ] */
                                x=z;
```

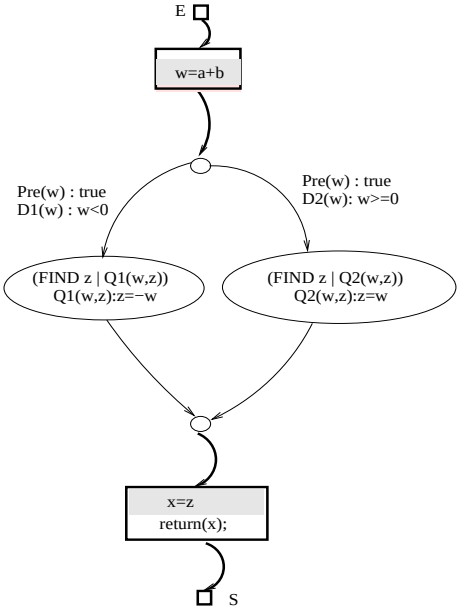

FIG. 11.8 – Graphe mixte de la fonction sous test

La figure 11.8 contient le résultat de la construction du graphe mixte correspondant à une fusion du graphe de flot de contrôle de la fonction sous test au graphe abstrait de la fonction imbriquée présentés dans la figure 11.7 avec une mise en correspondance des variables concrètes de l'appelant et des variables abstraites de la fonction imbriquée.

Dans la figure 11.8, nous avons grisé les instructions supplémentaires correspondant à cette mise en correspondance des variables.

La première mise en correspondance consiste à affecter aux variables d'entrée, représentées par W, les valeurs des expressions concrètes effectives d'appel. Pour la figure 11.8, cela correspond à l'ajout de l'égalité $w = a + b$. La seconde mise en correspondance correspond quant à elle à l'affectation des valeurs des variables abstraites de sorties Z aux variables concrètes de la fonction appelante. Pour la figure 11.8, cela correspond à l'ajout de l'égalité $x = z$. Dans la représentation des graphes mixtes, nous mettons aussi en relief le graphe de flot de contrôle de la fonction appelante (par des flèches en gras), graphe que nous cherchons à couvrir selon notre critère structurel.

11.3.2 Besoin de concrétisation de l'interface

Dans la précédente section, le fait que la fonction imbriquée ne prenne qu'une seule valeur en entrée et retourne qu'une seule valeur en sortie n'en fait pas une illustration très représentative de la construction de notre graphe mixte. Dans la plupart des cas, il peut y avoir plusieurs entrées et plusieurs sorties et la mise en correspondance des variables concrètes et abstraites requiert

plus d'efforts. Pour cela, il est nécessaire que l'utilisateur fournisse une interface concrète précisant les relations des variables structurelles et fonctionnelles des fonctions imbriquées. Autrement dit, la mise en œuvre de notre approche requiert des informations supplémentaires à l'utilisateur qui nécessite la mise en place d'un format dédié (éventuellement associé à des mécanismes de vérification).

Démonstration du besoin de caractérisation via un exemple

Prenons la figure 11.9 contenant la fonction `f` sous test appelant la fonction `div` calculant la division de deux termes passés en entrée et retournant le quotient entier de cette division ainsi que le reste de celle-ci. En entrée, la variable `x1` contient la valeur du numérateur, `*x2` la valeur du dénominateur et en sortie la valeur retournée correspond à `x1/*x2` et le reste de la division entière est contenu dans `*x2`.

```
1  int div(int x1,int * x2)
2  {
3    int result ;
4    result = x1/(*x2);
5    *x2=x1%(*x2);
6    return (result);
7  }
8  int f(int a, int b)
9  {
10  int r,q;
11  r=b;
12  q=div(a,&r); /*passage de r par adresse et de a par valeur*/
13  return(q);
14 }
```

FIG. 11.9 – Fonction `f` sous test appelant la fonction div

Le passage d'argument par adresse permet ici de retourner plus d'une sortie pour la fonction imbriquée. La spécification associée précise que cette fonction n'est pas définie pour des valeurs en entrée négatives. La variable d'entrée $p_2(W) = w_2$ représente le numérateur, $p_1(W) = w_1$ le dénominateur, $p_1(Z) = z_1$ le reste de la division entière et $p_2(Z) = z_2$ le résultat de la division entière.

$$Spec(div, W, Z) = \{PP_1(div, W, Z)\}$$

$$PP_1(div, W, Z) = (Pre(div, W) \wedge D_1(g, W), Q_1(g, W, Z)) =$$

$$((w_2 \geq 0 \wedge w_1 > 0) \wedge true, w_2 = z_1 + (z_2 * w_1) \wedge z_1 < w_1)$$

Nous mettons ici l'accent sur les difficultés de mise en correspondances des variables du graphe de flot de contrôle de l'appelant et des variables abstraites de la fonction imbriquée présentés dans la figure 11.10 pour la construction du graphe mixte de la fonction appelante.

D'après la spécification, nous savons que la fonction prend deux entrées représentées par w_1 et w_2 telles que $W = (w_1, w_2)$ et deux sorties z_1 et z_2 telles que $Z = (z_1, z_2)$. Le code source de la fonction `div` nous permet de définir les deux paramètres formels tels que $\mathcal{P}_{form} = [x1, x2]$ et l'analyse du code source de la fonction appelante, nous permet de définir les paramètres effectifs de l'unique occurrence d'appel : $\mathcal{P}_{eff} = [a, \&r]$.

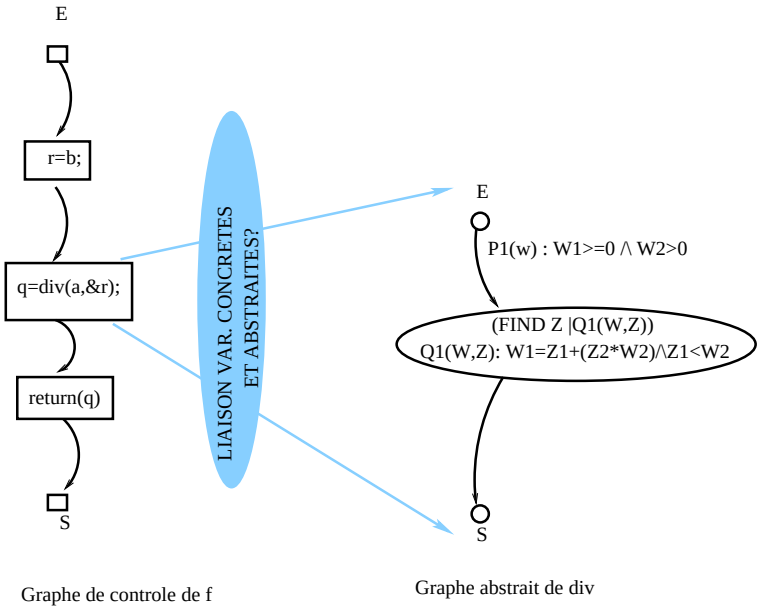

FIG. 11.10 – Graphe de contrôle de `f` et graphe abstrait de `div`

Nous avons donc deux possibilités pour initialiser W à savoir :

```
w_1=a;          /* OU */        w_1=r;
w_2=r;                          w_2=a;
```

Toujours grâce à l'analyse du code source on peut établir que la variable `q` reçoit une des sorties de la fonction imbriquée et qu'une autre est affectée à la variable `r`, passée par adresse.

Deux possibilités différentes s'offrent encore à nous pour récupérer les valeurs de sortie de notre fonction imbriquée à savoir :

```
q=z_1;          /* OU */        r=z_1;
r=z_2;                          q=z_2;
```

Nous devons donc définir parmi ces quatre possibilités, celles valides afin de pouvoir construire notre graphe mixte.

Concrétisation de l'interface

Notation 21
Nous désignons l'implantation d'une fonction `g` *par* `code(g)`.

L'utilisateur nous fournit l'interface associée au code source et à la spécification d'une fonction imbriquée. Ainsi une fonction `g` imbriquée est modélisée de la façon suivante :

$$Fct(\texttt{code(g)}, Spec(g, W, Z), \mathcal{I}(g))$$

avec $\mathcal{I}(g)$ l'interface fournie par l'utilisateur caractérisant le lien entre les variables concrètes et abstraites de la fonction imbriquée. Cette interface est indépendante des différentes occurrences d'appel de cette fonction imbriquée car elle dépend uniquement du couple (`code(g)`,$Spec(g, W, Z)$).

Nous disposons de plusieurs informations non liées à l'occurrence d'appel utiles pour la mise en correspondance des variables à savoir :

- $\mathcal{P}_{form}$ est la liste ordonnée des paramètres formels de la fonction imbriquée récupérée lors de l'analyse du code source de celle-ci,
- $VarGlob$ la liste des variables globales pouvant être utilisées et/ou définies dans g,
- W fourni lors de la spécification de la fonction est le vecteur de n composantes représentant les n entrées de la fonction dans la spécification et
- Z aussi fourni lors de la spécification de la fonction est le vecteur de p composantes représentant les p sorties de la fonction dans la spécification.

De plus, pour la i^{eme} occurrence statique d'appel, nous pouvons par analyse du code définir :

- $\mathcal{P}_{eff_i}$ est la liste ordonnée des paramètres effectifs de la fonction imbriquée pour une occurrence donnée ici la i^{eme} par convention de notation et
- R_i l'éventuelle variable de la fonction appelante récupérant la valeur de retour de la fonction g pour la i^{eme} occurrence d'appel. Rappelons, qu'après prétraitement, chaque instruction contient au plus une expression à effets de bord (cf. figures 11.1 et 11.2). Ainsi, la variable de R_i ne peut prendre que la valeur de retour d'un appel de fonction et non une expression de cette valeur de retour.

Rappelons ici que le nombre d'éléments de $\mathcal{P}_{form}$ est identique à celui de $\mathcal{P}_{eff_i}$ tel que :

$$|\mathcal{P}_{form}| = |\mathcal{P}_{eff_i}|$$

et aussi que le domaine du k^{eme} élément de chacune de ses listes est identique ce qui correspond à un même typage à savoir :

$$Dom(elem(\mathcal{P}_{form}, k)) = Dom(elem(\mathcal{P}_{eff_i}))$$

le k^{eme} élément de $\mathcal{P}_{form}$ prend la valeur de l'expression correspondant au k^{eme} élément de $\mathcal{P}_{eff_i}$ pour cette occurrence d'appel.

Convention 13
Lorsque la fonction imbriquée est une fonction C et non une procédure dans le sens où elle retourne une valeur, nous introduisons une nouvelle variable concrète pour représenter la valeur de retour de la fonction. Par convention, nous noterons cette variable $return\_\_\_fonc$ *pour une fonction avec retour nommée* $fonc$.
L'ensemble précédent noté $Ret(g)$ *représente soit l'ensemble vide* $\emptyset$ *pour le cas d'une procédure* g *ou pour une fonction* g *l'ensemble à un élément tel que :* $Ret(g) = \{return\_\_\_g\}$.

Une occurrence d'appel est ainsi caractérisée par le couple (R_i,$\mathcal{P}_{eff_i}$). L'interface $\mathcal{I}(g)$ relie les variables statiques abstraites (W, Z) aux variables concrètes de $\mathcal{P}_{form}$, $VarGlob$ et $Ret(g)$.

L'interface $\mathcal{I}$ est une application de profil :

$$\mathcal{I} : W \cup Z \rightarrow (VarGlob \cup \mathcal{P}_{form} \cup Ret(g)))$$

Notons que seul un sous-ensemble des paramètres formels et des variables globales représentant des entrées et/ou sorties de la fonction est utilisé. En effet, un paramètre formel peut ne pas être défini ni utilisé dans la fonction auquel il appartient et de même pour les variables globales du programme auquel appartient la fonction g.

L'interface $\mathcal{I}$ est éventuellement représentable par une liste de $n+p$ couples de la forme suivante :

$$\mathcal{I} = \{(w_1, v_1), ..., (w_n, v_n), (z_1, vv_1), ..., (z_p, vv_p)\}$$

avec

$$\mathcal{I}(w_i)_{i\in[1...n]} = v_i$$
$$\mathcal{I}(z_j)_{i\in[1...p]} = vv_p$$

où $v_1, ..., v_n, vv_1, , ..., vv_p \in (VarGlob \cup \mathcal{P}_{form} \cup Ret(g))$ avec la possibilité qu'une de ses variables soit présente dans différents couples de $\mathcal{I}$.

Précisons la dualité existante entre les entrées W et les sorties Z abstraites de la fonction. Les variables d'entrées abstraites d'une fonction imbriquée sont affectées par les valeurs des expressions des variables concrètes de la fonction sous test et après détermination des sorties abstraites, un sous-ensemble des variables concrètes de la fonction sous test sont affectées des valeurs des variables abstraites de sortie selon les informations fournies par l'interface.

Reprenons ainsi l'exemple de l'appel de la fonction `div` de la figure 11.9. L'interface fournie est :

$$\mathcal{I} = \{(w_1, *x2), (w_2, x1), (z_1, *x2), (z_2, g\_\_\_result)\}$$

Pour cette occurrence, nous avons $\mathcal{P}_{eff} = [a, \&r]$ et $R = q$. Soient $\mathcal{P}_{form} = [x1, *x2]$ et $Ret(g) = g\_\_\_result$, les liens entre $\mathcal{P}_{form}$ et $\mathcal{P}_{eff}$ ainsi qu'entre $Ret(g)$ et R.

Cela nous permet d'établir la mise en correspondance exacte entre les variables concrètes et abstraites de la fonction imbriquée à savoir :

```
w1=r; w2=a;
r=z1; q=z2;
```

Illustration 79
Dans l'interface la variable w1 est rattachée à la variable `*x2` *deuxième élément de* $\mathcal{P}_{form}$ *: w1 prend donc la valeur de la seconde expression effective d'appel* $\mathcal{P}_{eff} = [a, \&r]$ *soit la valeur de la variable* `r`.

Nous pouvons alors construire le graphe mixte de la fonction `f` (cf. figure 11.11).

Modifions maintenant la fonction sous test pour voir le cas d'une occurrence d'appel plus complexe. Pour cela, nous modifions l'instruction d'appel de la figure 11.9 de la façon suivante :

```
q=div(a,&r);          /* devient */          q=div(a*a,&r) +1;
```

Comme nous l'avons déjà dit précédemment, le code est tout d'abord soumis au prétraitement de façon à ce qu'une instruction séquentielle possède au plus une instruction d'appel. Ainsi l'instruction contenant l'appel est transformée de la façon suivante :

```
q=div(a*a,&r) +1;        /* => */        tmp= div(a*a,&r);
                                         q=tmp+1;
```

Ainsi, la variable de l'occurrence d'appel $\mathcal{R}_i$ correspondra toujours à la variable éventuelle de $Ret(g)$ et en aucun cas à une expression sur cette même variable.

Lors de l'analyse du code source de la fonction imbriquée, nous allons déterminer que $\mathcal{R}_i = \{tmp\}$ et que $\mathcal{P}_{eff_i} = [a * a, r]$. En utilisant toujours la même interface $\mathcal{I} = \{(w_1, *x2), (w_2, x1), (z_1, *x2), (z_2, g\_\_\_result)\}$, nous déduisons les informations suivantes :

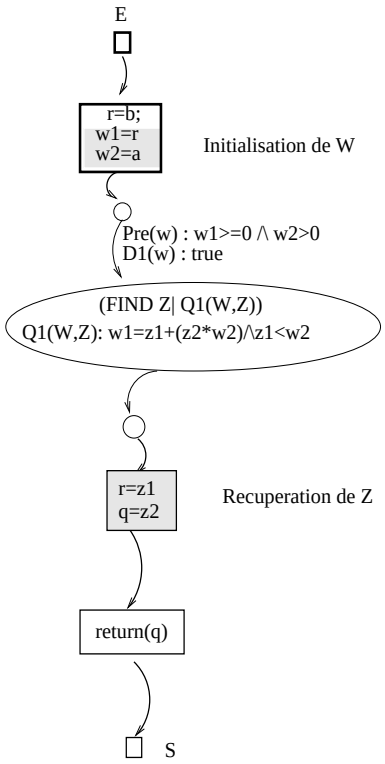

FIG. 11.11 – Graphe mixte de la fonction f appelant la fonction div

- w_1 est associé à la variable `*x2` correspondant à $elem(\mathcal{P}_{form}, 2)$, lui-même associé à $elem(\mathcal{P}_{eff_i}, 2)$ soit la variable `r`,
- w_2 est associé à la variable `x1` correspondant à $elem(\mathcal{P}_{form}, 1)$, lui-même associé à $elem(\mathcal{P}_{eff_i}, 1)$ soit l'expression `a*a`,
- z_1 est associé à la variable `*x2` correspondant à $elem(\mathcal{P}_{form}, 2)$, lui-même associé à $elem(\mathcal{P}_{eff_i}, 2)$ soit la variable `r`,
- z_2 est associé à la variable `g___result` correspondant à $Ret(g)$, lui-même associé à R_i soit la variable `tmp`.

La mise en correspondance des variables abstraites et concrètes de la fonction imbriquées est ainsi :

```
w_1=r; w_2=a*a;
r=z_1; tmp=z_2;
```

11.3.3 Différentes types des variables d'entrée et de sortie d'une fonction C

Comme nous l'avons vu dans les exemples précédents, il existe différentes façons pour passer une variable comme argument (par valeur ou par adresse) d'une fonction et aussi pour conserver/-récupérer les modifications opérées dans une fonction (retour de fonction, passage par adresse). Il existe une autre possibilité : l'utilisation de variables globales dont la valeur peut servir en entrée et en sortie de la fonction imbriquée car toute modification est conservée en dehors des appels de fonction. Nous allons ici nous attarder sur ces différentes façons de passer une variable comme entrée et/ou sortie d'une fonction imbriquée et les conséquences induites par rapport à l'interface de la fonction imbriquée.

11.3.4 Conséquence sur l'interface

Le passage par valeur d'argument dans une fonction correspond à créer une variable locale dans la fonction imbriquée de la valeur initiale passée par argument. Toute modification faite à ce paramètre est locale à la fonction imbriquée et par conséquent ce paramètre ne peut faire partie que des variables en entrée de la fonction et non de ses variables en sortie. Une variable passée comme argument par valeur peut être associée uniquement à une des variables d'entrée de la spécification.

Le retour de fonction représente la valeur retournée par une fonction à la fin de son exécution. Cette valeur est une valeur de sortie généralement récupérée par une variable de la fonction sous test qui peut donc n'être associée qu'à une variable de sortie de la spécification par l'interface fournie par l'utilisateur.

Lors d'un passage par adresse d'une variable, toute modification faite dans la fonction est directement faite au contenu de l'adresse mémoire de la variable passée par adresse. Par conséquent, toute modification faite dans le corps d'une fonction est conservée à la sortie de la dite fonction pour toutes les variables passées par adresse. Ainsi une variable passée comme argument par adresse peut être associée à une variable d'entrée et/ou à une variable de sortie de la spécification de la fonction imbriquée.

Enfin, une variable globale est une variable accessible en lecture et écriture dans toute fonction appartenant au même fichier dans lequel elle est déclarée. Cela signifie que toute modification faite dans une fonction est conservée à sa sortie. Une variable globale peut également être associée à une variable en entrée et/ou en sortie de la spécification.

11.4 Construction du graphe mixte d'une fonction sous test

En remplaçant tous les blocs d'appel dans le CFG de la fonction sous test par les graphes abstraits des fonctions imbriquées associées, graphes précédés et succédés des affectations de mise en correspondance des variables concrètes et abstraites, nous obtenons ce que nous définissons comme le graphe mixte de la fonction sous test.

Une instruction d'appel est remplacée par une séquence d'affectations et de meta-instruction basées sur la spécification et l'interface associée de la fonction imbriquée. Nous désignerons par $call(g(i))$ l'instruction de la i^{eme} occurrence d'appel de la procédure ou fonction imbriquée `g`. Nous désignerons aussi par `code(f)` le code source de la fonction appelante.

Commençons tout d'abord par donner l'algorithme général de substitution.

BEGIN SUBSTITUTION APPEL
ENTRÉES : $code(f), Spec(g, W, Z), \mathcal{P}_{form}, \mathcal{I}(g), Ret(g)$
Pour chaque $call(g(i))$ dans `code(f)` faire :
    déterminer R_i et $\mathcal{P}_{eff_i}$
    supprimer instruction $call(g(i))$
    PRE-APPEL$(\mathcal{I}(g), \mathcal{P}_{form}, \mathcal{P}_{eff_i})$
    META-INSTRUCTIONS$(Spec(g, W, Z))$
    POST-APPEL$(\mathcal{I}(g), \mathcal{P}_{form}, \mathcal{P}_{eff_i}, R_i, Ret(g))$
FinPour
END SUBSTITUTION APPEL

Cet algorithme fait appel à un premier algorithme PRE-APPEL qui a pour objectif l'initialisation des variables abstraites d'entrée W par les valeurs des expressions effectives des variables concrètes de la fonction imbriquée.

BEGIN PRE-APPEL
ENTRÉES : $\mathcal{P}_{form}, \mathcal{I}(g), \mathcal{P}_{eff_i}$
Pour les n couples $(w_k, v) \in \mathcal{I}(g)$ faire :
    Si $v \in \mathcal{P}_{form}$ tel que $elem(\mathcal{P}_{form}, r) = v$ faire :
        $w_k = elem(\mathcal{P}_{eff_i}, r)$
    Sinon faire :
        /* cas d'une variable globale */
        $w_k = v$
    FinSi
FinPour
END PRE-APPEL

De même, l'algorithme POST-APPEL permet d'affecter à un sous-ensemble des variables concrètes de la fonction sous test les valeurs des sorties abstraites Z.

BEGIN POST-APPEL
ENTRÉES : $\mathcal{P}_{form}, \mathcal{I}(g), \mathcal{P}_{eff_i}, R_i, Ret(g)$
Pour les p couples $(z_l, vv) \in \mathcal{I}$ faire :
    Si $vv \in \mathcal{P}_{form}$ tel que $elem(\mathcal{P}_{form}, r) = vv$ faire :
        $elem(\mathcal{P}_{eff_i}, r) = z_l$
    Sinon faire :
        Si $vv \in Ret(g)$ alors faire :
            Si $R_i \neq \emptyset$ alors faire :
                $elem(R_i, 1) = z_l$
            /*le cas contraire signifie que le retour de fonction est non utilisé */
            FinSi
        Sinon /* cas d'une variable globale */ faire :
            $vv = z_l$
        FinSi
    FinSi
FinPour
END POST-APPEL

Enfin, il nous reste à insérer les meta-instructions issues de la spécification de la fonction.

BEGIN META-INSTRUCTIONS
ENTRÉES :$Spec(fonc, W, Z)$
Écrire instruction de choix Case sur W
Pour chaque couple $(Pre(fonc, W) \wedge D_i(fonc, W), Q_i(fonc, W, Z)) \in Spec(fonc, W, Z)$
    Traduire les contraintes de $Pre(fonc, W) \wedge D_i(fonc, W)$ en conditions du Case
    Implanter le prédicat existentiel $(FIND\ Z | Q_i(fonc, W, Z))$ en instructions du Case
FinPour
END META-INSTRUCTIONS

En ce qui concerne l'ordre des affectations des variables abstraites d'entrée et des variables concrètes par les valeurs des variables abstraites de sortie, nous prenons en compte l'ordre d'affectation du langage C. Notons que nous excluons le cas comportant une relation d'alias entre les variables d'entrée de la fonction appelée.

Ce graphe mixte d'une fonction sous test peut être manipulé comme tout autre graphe de flot de contrôle : nous pouvons le couvrir selon un critère de test structurel habituel.

Une telle modélisation permet ainsi de conserver une méthode structurelle de test avec couverture de graphe tout en abstrayant les chemins structurels des fonctions imbriquées par des meta-instructions fonctionnelles. Ces meta-instructions étant issues des spécifications des fonc-

tions imbriquées, le comportement réel de ces fonctions est donc pris en compte avec un niveau supérieur d'abstraction ce qui permet de limiter l'explosion combinatoire des chemins induite d'un traitement des fonctions imbriquées selon une stratégie "inlining". Le nombre de chemins à couvrir est donc diminué tout en maintenant la couverture structurelle désirée de la fonction sous test.

Le chapitre suivant réutilise notre modélisation des chemins structurels de la fonction sous test avec appels pour l'appliquer à notre méthode de test PathCrawler.

Chapitre 12

Soumission du graphe mixte d'une fonction sous test à la méthode PathCrawler

Dans ce chapitre nous allons reprendre la stratégie de sélection des cas de test de la méthode PathCrawler présentée dans le chapitre 7. Nous n'allons plus appliquer cette stratégie sur le graphe de flot de contrôle de la fonction sous test mais sur le graphe mixte d'une fonction sous test contenant des appels de fonctions selon la modélisation expliquée dans le chapitre 11. Nous définirons ainsi un nouveau critère de test approprié à la nouvelle modélisation des fonctions sous test avec appels.

Dans une première section, nous verrons l'application de ce nouveau critère de test sur le graphe mixte de la fonction sous test et nous discuterons de la couverture de test en découlant.

Dans la section suivante, nous définirons un nouveau critère de test dédié à la nouvelle modélisation des fonctions sous test avec appels permettant d'éviter une redondance dans la couverture des chemins structurels de la fonction sous test.

12.1 Contexte

Initialement, dans la méthode de test unitaire PathCrawler, les fonctions imbriquées sont soumises à un traitement "inlining" ajoutant la combinatoire des chemins des fonctions imbriquées à la combinatoire des chemins de la fonction sous test comme le montre la figure 12.1.

Nous avons présenté dans le chapitre 11 précédent une nouvelle modélisation des chemins structurels contenant des instructions d'appel. Par une utilisation de la spécification exprimée sous forme de couples pre/post, les chemins internes des fonctions imbriquées sont abstraits par l'expression du domaine fonctionnel imbriqué correspondant.
Nous conservons les codes sources (ou exécutables) des fonctions imbriquées qui sont exécutés lors des différents cas de test de la fonction sous test ce qui nous permet de conserver la réelle exécution de la fonction sous test. C'est lors du calcul du prédicat de chemin du cas de test courant que les chemins internes aux fonctions imbriquées sont abstraits par leur expression fonctionnelle ce qui nous amène à substituer dans le CFG de la fonction les blocs d'appel par leur graphes abstraits construits à partir de leur spécification avec une mise en correspondance des variables structurelles et fonctionnelles. Nous obtenons ainsi ce que nous définissons comme le graphe mixte de la fonction sous test (cf. figure 12.2).

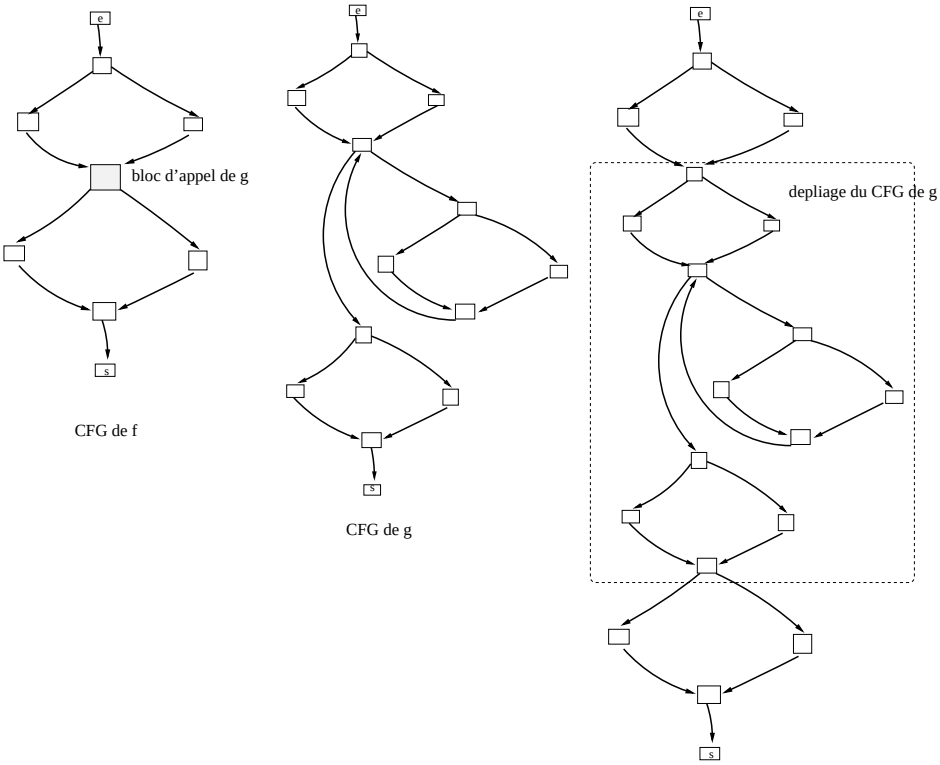

FIG. 12.1 – Illustration du dépliage de la fonction g appelée dans la fonction `f`

C'est l'utilisateur qui fournit la spécification des fonctions imbriquées et nous lui demandons de respecter les différentes contraintes imposées. Il est cependant possible qu'une ou plusieurs de

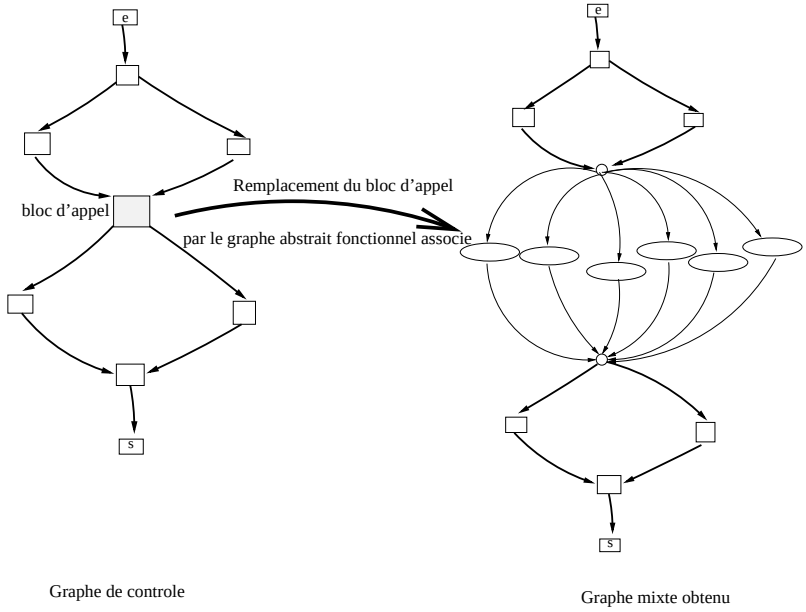

FIG. 12.2 – Du graphe de contrôle au graphe mixte de la fonction sous test

ces contraintes ne soient pas respectées.

En cas de spécification non complète, deux traitements sont envisageables :

- nous pouvons proposer à l'utilisateur soit un traitement "inlining" de la fonction concernée sur tout son domaine de définition,
- soit un traitement "inlining" uniquement pour le sous-domaine d'entrée non spécifié $Def(g) \setminus Dom(Spec(g, W, Z))$.

De même, pour une spécification non déterministe, plusieurs instanciations de Z peuvent correspondre à une unique instanciation de W. Nous pouvons choisir de conserver la première instanciation de Z puis de passer à un autre cas de test. Dans ce cas, nous ne pouvons pas garantir le maintien de la couverture des chemins de la fonction sous test. Pour être surs de ne perdre aucun chemin structurel de la fonction sous test, il faut forcer toutes les instanciations possibles de Z. Cela peut entraîner une redondance entre les différents cas de test.

Pour des domaines fonctionnels non exclusifs, deux cas sont à distinguer selon que les postconditions associées soient ou non contradictoires. Si les postconditions concordent (les mêmes sorties sont obtenues) alors nous perdons en efficacité car nous pouvons couvrir deux fois le même domaine de la fonction imbriquée à savoir le domaine commun entre deux domaines fonctionnels non exclusifs. C'est le scénario le moins ennuyeux même si cela peu entraîner une redondance des tests. En revanche, si les postconditions sont contradictoires (différentes sorties sont obtenues), nous revenons au cas d'une spécification non déterministe.

Dans la section suivante, nous allons utiliser ce graphe mixte comme le graphe de flot de contrôle de la fonction sous test et le soumettre à la stratégie de PathCrawler.

12.2 Encodage du graphe abstrait dans PathCrawler

Comme pour la méthode unitaire PathCrawler, l'encodage en contraintes se fait lors de la construction du chemin parcouru pour le cas de test courant. Comme pour les instructions de trace, la trace fonctionnelle issue d'un graphe abstrait et donc de la spécification d'une fonction imbriquée est encodé au moment de la construction du chemin et de la détermination du prédicat de chemin associé. Les contraintes encodées à partir du graphe abstrait d'une fonction imbriquée comme les contraintes encodées à l'instrumentation font parti des deux systèmes de contraintes manipulés : le prédicat de chemin et les contraintes du prochain domaine de sélection des cas de test.

Il est important de préciser que, jusqu'à présent, c'est-à-dire sans graphe abstrait, les systèmes de contraintes manipulés dans PathCrawler sont uniquement des conjonctions de conditions simples rencontrées le long d'un chemin d'exécution donné. Ceci est dû à la décomposition des conditions multiples lors du prétraitement des fonctions sous test.

Deux cas sont à distinguer :

- l'encodage commun avec l'instrumentation des fonctions dans la méthode PathCrawler et
- l'encodage propre au graphe abstrait du langage de spécification.

12.2.1 Contraintes arithmétiques : encodage dans la continuité de PathCrawler

Les contraintes issues de la spécification correspondant à des contraintes arithmétiques restent dans la continuité de l'encodage vu pour l'instrumentation des fonctions dans le chapitre 7 présentant la méthode PathCrawler. Des contraintes arithmétiques sont des contraintes d'égalité, de différence ou d'inégalité entre des expressions arithmétiques sur les domaines finis. Une expression arithmétique sur les domaines finis est une expression arithmétique utilisant les opérateurs classiques (+, -, *, /, etc) et dont les opérandes sont des variables, d'autres expressions arithmétiques ou des constantes.

12.2.2 Autres contraintes

La difficulté d'encodage concerne la richesse du langage de spécification présenté dans le chapitre 3 à savoir les quantificateurs universel et existentiel, la disjonction et la négation.

Le quantificateur existentiel sur domaine fini correspond à une disjonction de base sur tous les éléments du domaine et de même la quantification universelle sur domaine fini à une conjonction sur tous les éléments du domaine.

Nous utilisons alors des contraintes booléennes. Une contrainte booléenne sur les domaines finis est une expression composée à partir des opérateurs booléens comme le "non logique", le "ou logique", le "et logique" ....

Le traitement le plus souvent observé des contraintes à connecteurs logique est la réification de ces contraintes. Les opérandes d'une contrainte booléenne peuvent être la valeur entière 0 (interprétée comme "faux"), la valeur entière 1 (interprétée comme "vrai"), une variable booléenne (dont le domaine est restreint aux valeurs 0 et 1) ou une contrainte booléenne. Une contrainte utilisée comme une opérande de contrainte booléenne est "réifiée" en la remplaçant par une variable booléenne représentant la valeur de cette contrainte. Dès que le solveur de contraintes peut déduire que cette contrainte est vraie alors la variable booléenne est instanciée à 1 et à 0 si le solveur de contraintes arrive à prouver quelle est fausse. Cette variable booléenne est ensuite remplacée dans le système de contraintes par sa contrainte d'origine de valeur connue maintenant.

Les disjonctions de contraintes réifiées sont considérées comme des contraintes comme les autres. Ces disjonctions connectent des variables booléennes issues de la réification des contraintes. Il est possible d'utiliser des versions réifiées des opérateurs de comparaison ($<$, $=$, etc.). Nous pouvons aussi utiliser des opérateurs booléens pour permettre d'exprimer des disjonctions ou négations de contraintes. Une variable booléenne contenant la valeur de vérité d'une contrainte réifiée peut être utilisée dans une quelconque combinaison de contraintes booléennes. Notons que les disjonctions sont plus faciles à prouver comme vraies et les conjonctions à réfuter.

L'implication se traduit et se manipule comme une disjonction. En effet, la formule $A \Rightarrow B$ est équivalente à la formule $\neg A \vee B$.

Enfin, pour la négation, il faut préciser les différentes règles de simplification comme le résultat de la négation sur un quantificateur, une conjonction, une négation etc...

Remarque(s) 33
L'efficacité de résolution des contraintes réifiées n'est certainement pas optimale. Ces contraintes réifiées pourront être remplacées par la suite par des contraintes globales comme pour le traitement de la structure conditionnelle IfThenElse dans INKA [Got00] ou encore comme dans [MA00].

12.2.3 Encodage du FIND

L'encodage du FIND est direct dans notre contexte. En effet, si une des branches de notre graphe abstrait a été exercée les contraintes associées $Pre(g, W)$ et $D_i(g, W)$ sont vérifiées ce qui permet de définir une instanciation pour W répondant également aux contraintes du chemin partiel précédant l'appel dans la fonction sous test. La soumission de ces contraintes et de $Q_i(g, W, Z)$ à notre solveur de contraintes correspond à un CSP dont la solution est de déterminer toutes les instanciations valides de Z. Du fait que nos spécifications soient complètes et déterministes, pour une instanciation donnée de W, il existe une unique instanciation de Z répondant au CSP.

12.3 Définition d'un nouveau critère de test pour la couverture du graphe mixte d'une fonction sous test

Nous soumettons donc le graphe mixte de la fonction sous test présenté dans la figure 12.3 à la stratégie PathCrawler dans le but de couvrir tous les chemins faisables du graphe mixte de la fonction sous test. Nous définissons un nouveau critère de test, le critère TLCM pour tous-les-chemins-mixtes. Pour l'exemple de la figure 12.3, notre graphe mixte contient en tout 24 chemins "structuro-fonctionnels" soit 24 chemins mixtes.

12.3.1 Critère TLCM

Définition – 12.3.1

*Le **critère TLCM** correspond à couvrir les chemins du graphe mixte de la fonction sous test correspondant à la couverture :*

- *de chaque chemin structurel faisable de la fonction sous test et*
- *de tous les domaines fonctionnels faisables des fonctions imbriquées pour chacun de leurs contextes d'appel.*

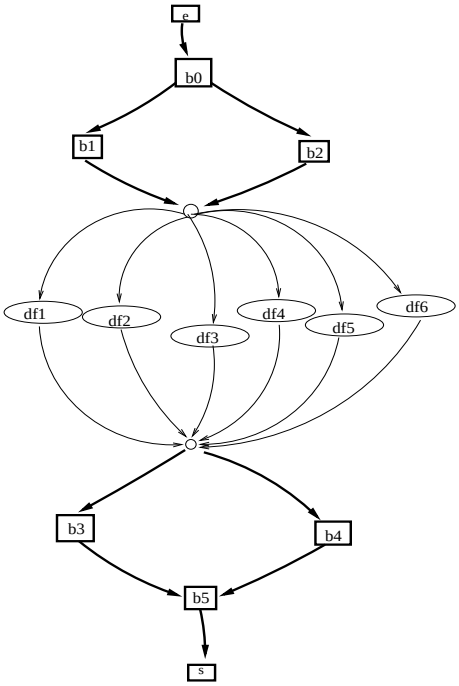

Fig. 12.3 – Graphe mixte de la fonction `f`

Nous raisonnons uniquement en terme de chemins couverts. Le graphe de contrôle de la fonction sous test est celui de la figure 12.2 composé de 4 chemins structurels. Nous supposons ici que tous les chemins du graphe mixte de la figure 12.3 sont faisables. Nous réutilisons les annotations des nœuds du graphe pour décrire les différents chemins couverts pendant les cas de test successifs.

Nous rappelons que la stratégie de sélection des cas de test de PathCrawler consiste à couvrir le graphe selon une stratégie en profondeur d'abord en niant la dernière contrainte du prédicat de chemin courant comme expliqué dans le chapitre 7.

Un premier cas de test sur le graphe mixte de la figure 12.3 nous amène à couvrir le chemin $(e, b0, b2, df2, b4, b5, s)$. Selon la stratégie PathCrawler, le second cas de test permet de couvrir le chemin $(e, b0, b2, df2, b3, b5, s)$. Pour le prochain cas de test, on remonte dans les contraintes du prédicat ce qui consiste à nier une contrainte interne à la fonction imbriquée c'est-à-dire de forcer le passage dans un autre domaine fonctionnel de la fonction imbriquée. Le chemin structurel couvert est alors $(e, b0, b2, df3, b3, b5, s)$. La couverture des chemins du graphe mixte s'arrête dès que tous les chemins faisables ont été couverts. La figure 12.4 illustre les premiers chemins couverts dans notre graphe mixte. Les chemins couverts sont mis en relief dans cette figure par un tracé en pointillés.

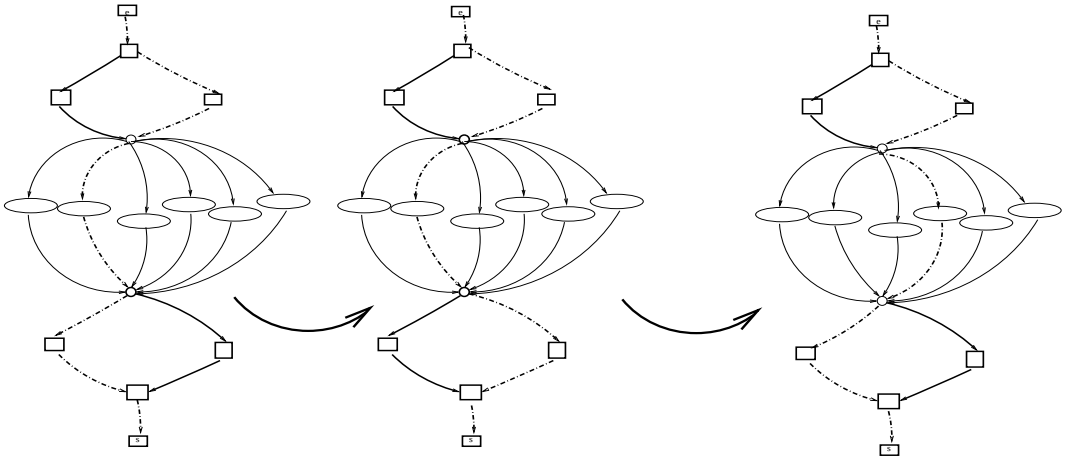

FIG. 12.4 – Chemins couverts lors de l'application du critère TLCM

12.3.2 Analyse

Couverture de la fonction sous test et de la fonction imbriquée

L'application du critère TLCM sur le graphe mixte de la fonction sous test selon la stratégie de sélection de PathCrawler consiste à couvrir tous les chemins faisables du graphe mixte et plus précisément en raisonnant en termes de chemins structurels et de chemins abstraits :

- tous les chemins partiels structurels de la fonction sous test en sortie de chaque chemin abstrait de la fonction imbriquée sont couverts,
- tous les chemins abstraits de la fonction imbriquée pour chaque chemin partiel structurel amenant à une instruction d'appel sont couverts.

Ainsi, d'une part, l'application du critère TLCM sur le graphe mixte de la fonction sous test amène à couvrir tous les chemins structurels faisables de la fonction sous test sont couverts ainsi que tous les chemins abstraits faisables de la fonction imbriquée pour chaque chemin partiel structurel amenant à l'appel de la fonction imbriqué. L'application du critère TLCM sur le graphe mixte de la fonction sous test consiste ainsi en une double couverture à savoir une couverture structurelle des chemins de la fonction sous test et une couverture fonctionnelle des domaines fonctionnels de la fonction imbriquée pour chacun de ses contextes d'appel.

Test en contexte de la fonction imbriquée

Nous rappelons que la fonction imbriquée est exécutée lors des différents cas de test et que l'abstraction de ses chemins internes est faite lors du calcul des prédicats de chemin comme expliqué dans le chapitre 11. Nous disposons des valeurs réelles en entrée et en sortie de la fonction imbriquée à chaque cas de test de la fonction sous test. La fonction imbriquée est exécutée dans son environnement réel (la fonction sous test) et pour chacun de ses contextes d'appel.

La spécification de la fonction imbriquée exprimée sous forme de couples pre/post $(Pre(f,X) \wedge D_i(f,X), Q_i(f,X,Y))$ permet de tester en contexte la fonction appelée : les valeurs en entrée de la fonction imbriquée vérifient les contraintes de la première partie $Pre(f,X) \wedge D_i(f,X)$ d'un couple pre/post de la spécification et ses valeurs en sortie doivent vérifier la seconde partie $Q_i(f,X,Y)$ du même couple pre/post.

Nous pouvons donc facilement et sans effort mettre en place une méthode de test en contexte de la fonction imbriquée. Étant dans une stratégie ascendante de test, les fonctions de bas niveaux (appelées) sont testées unitairement avant les fonctions de plus haut niveau (appelantes). Disposant du code source (ou de l'exécutable) des fonctions imbriquées et la fonction imbriquée ayant été testée unitairement et validée par rapport à ses spécifications via l'oracle, nous pouvons mettre en place une technique de test en contexte pour ces fonctions. Une idée serait ici de réutiliser

la spécification des fonctions imbriquées comme oracle dans le contexte c'est-à-dire de vérifier à chaque exécution que les entrées et sorties réelles de la fonction imbriquée vérifient bien les spécifications et donc la relation entrées/sorties spécifiée. $Q_i(f, X, Y)$ jouerait le rôle d'oracle pour le domaine en entrée de la fonction caractérisé par $Pre(f, X) \wedge D_i(f, X)$. Cela permettrait également de vérifier l'absence de biais dans la couverture de la fonction appelante c'est-à-dire pas d'utilisation des fonctions imbriquées en dehors de leurs domaines de définition.

Remarque(s) 34
De façon générale, en disposant du code source d'une fonction et d'une spécification formalisée, il est possible de couvrir structurellement cette fonction en utilisant la spécification pour la mise en place d'un oracle.

On pourrait ainsi augmenter la confiance dans la fonction sous test en lui appliquant une méthode de test fonctionnel dans le contexte. Cela permettrait d'identifier des utilisations des fonctions imbriquées hors domaine, défaut couramment détecté lors du test d'intégration et du test en contexte. De plus, nous pourrions également lever une de nos hypothèses à savoir le test préalable unitaire des fonctions appelées. En effet, nous pourrions tester parallèlement la fonction sous test structurellement et les fonctions imbriquées fonctionnellement dans le contexte.

Redondance dans la couverture des chemins structurels de la fonction sous test

Lors de l'application de la méthode PathCrawler sur le graphe mixte de la fonction sous test de la figure 12.3, les chemins couverts lors des trois premiers cas de test sont :
- $(e, b0, b2, df2, b4, b5, s)$,
- $(e, b0, b2, df2, b3, b5, s)$ et
- $(e, b0, b2, df3, b3, b5, s)$ (cf. figure 12.4).

Si nous nous concentrons uniquement sur les chemins structurels de la fonction sous test, nous remarquons que ceux-ci sont couverts plusieurs fois. Par exemple le second et le troisième cas de test de notre graphe mixte couvre le même chemin structurel à savoir $(e, b0, b2, b3, b5, s)$.

L'application du critère TLCM sur le graphe mixte de la fonction sous test peut entraîner une redondance dans la couverture structurelle des chemins de la fonction sous test. Toujours en supposant que tous les chemins du graphe mixte de la figure 12.3 sont faisables, nous couvrons donc ses 24 chemins alors que notre objectif premier est de couvrir uniquement les 4 chemins structurels de la fonction sous test.

Nous allons donc proposer dans la section suivante un nouveau critère de test à appliquer au graphe mixte de la fonction sous test afin d'éviter cette redondance de tests.

12.4 Autre critère de couverture du graphe mixte d'une fonction sous test

En nous concentrant uniquement sur notre objectif premier qui est la couverture de tous les chemins faisables de la fonction sous test, nous écartons donc l'éventuel test en contexte des fonctions imbriquées pour nous orienter uniquement sur la couverture structurelle des chemins des fonctions sous test avec instructions d'appel en éliminant les redondances entre les différents cas de test effectués.

12.4.1 Critère TLCS

Définition – 12.4.1

> *Le **critère TLCS** correspond à couvrir les chemins du graphe mixte de la fonction sous test correspondant à la couverture de chaque chemin structurel faisable de la fonction sous test.*

L'exploration systèmatique de tous les domaines fonctionnels de la fonction imbriquée nous permet de garantir le maintien de la couverture de 100% des chemins faisables de la fonction sous test. Cela s'explique facilement dans la mesure où un chemin partiel en sortie de la fonction imbriqué peut-être non exécutable pour un domaine fonctionnel donné de la fonction imbriqué et exécutable pour un autre de ses domaines fonctionnels. Le fait de forcer le passage dans tous les domaines fonctionnels nous garantit donc de couvrir tous les chemins partiels exécutables en sortie de la fonction imbriquées.

Cependant, nous venons de voir, dans la section précédente que la négation systématique des domaines fonctionnels des fonctions imbriquées introduit une redondance dans la couverture des chemins de la fonction sous test.

L'idée est donc de nier un domaine fonctionnel d'une fonction imbriquée uniquement si il existe un chemin partiel en sortie de cette fonction qui n'a pas pu être couvert et ce, afin de trouver un domaine fonctionnel de la fonction imbriquée (s'il en existe un) permettant de couvrir ce chemin partiel encore non couvert. Nous définissons ainsi le nouveau critère TCLS pour tous-les-chemins-structurels correspondant à une restriction du précédent critère TLCM correspondant à la couverture unique des chemins structurels de la fonctions sous test éliminant toute redondance entre les différents cas de test.

12.4.2 Couverture du graphe mixte d'une fonction sous test pour le critère TLCS

L'application du critère TLCS sur le graphe mixte d'une fonction sous test correspond à forcer le passage dans un nouveau domaine fonctionnel de la fonction imbriquée pour un contexte d'appel donné uniquement s'il existe un chemin partiel structurel en sortie de la fonction imbriquée n'ayant pas encore été couvert. Ainsi, si tous les chemins partiels structurels en sortie ont été couverts, la stratégie procède à la négation de la contrainte du prédicat de chemin précédant l'appel de la fonction imbriquée par backtrack.

Nous reprenons le graphe mixte de la fonction sous test précédent de la figure 12.3.

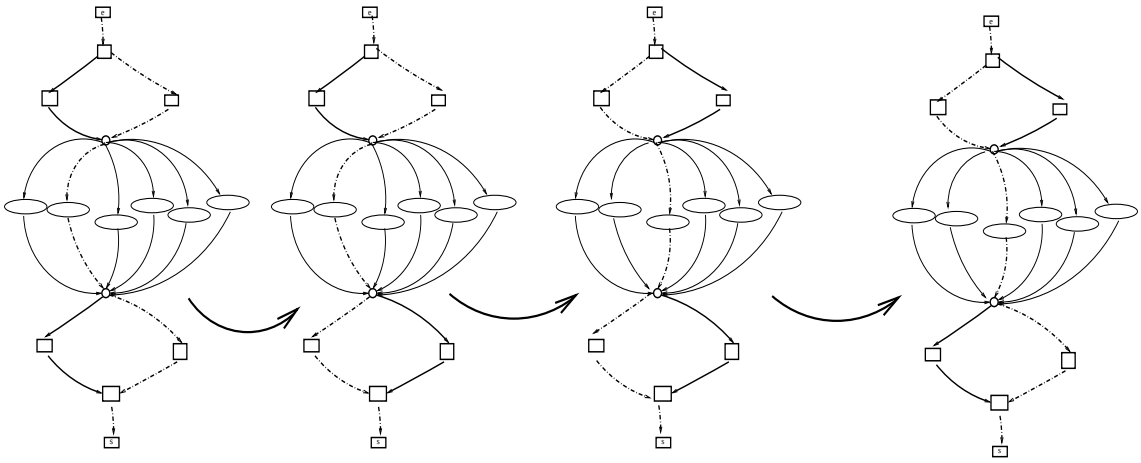

Fig. 12.5 – Chemins couverts lors de l'application du critère TLCS

Nous reprenons le premier cas de test de la section précédente couvrant le chemin

$(e, b0, b2, df2, b4, b5, s)$. Selon la stratégie PathCrawler, la dernière contrainte du prédicat de chemin correspondant à une contrainte de la fonction sous test est niée et donc le chemin $(e, b0, b2, df2, b3, b5, s)$ est couvert. Pour le prochain cas de test, la stratégie amène à nier une contrainte interne à la fonction imbriquée. Tous les chemins partiels en sortie de la fonction imbriquée ayant déjà été couverts par les cas de test précédents, la négation du domaine fonctionnel de la fonction imbriquée est inutile. La stratégie procède à un backtrack consistant à nier la contrainte précédant l'appel de la fonction. Le troisième cas de test couvre ainsi le chemin $(e, b0, b1, df3, b3, b5, s)$. Le quatrième et dernier cas de test couvre alors le chemin $(e, b0, b1, df3, b4, b5, s)$. Tous les chemins structurels de la fonction sous test ont été couverts, le test s'arrête donc (cf. figure 12.5).

Nous identifions pour chaque contexte d'appel d'une fonction imbriquée l'ensemble des les chemins en sortie de la fonction encore non couverts. Lorsque nous forçons un nouveau domaine fonctionnel de la fonction imbriquée, nous forçons chacun des chemins en sortie encore non exécutés pour le contexte d'appel en cours. Ainsi, nous ne couvrons pas de chemin en sortie de la fonction imbriqué qui aurait déjà été couvert par un domaine fonctionnel précédemment activé. Par l'application du critère TLCS au graphe mixte des fonctions sous test, nous maintenons la couverture de 100% des chemins de la fonction sous test (les 4 chemins ont bien été couverts) en enlevant une redondance dans les tests des chemins structurels de la fonction sous test (4 cas de test ont été suffisants).

12.5 Conclusion

Nous avons décrit dans ce chapitre la soumission du graphe mixte d'une fonction sous test au cas particulier de la méthode de test PathCrawler. Notons que notre modélisation du graphe mixte des fonctions sous test peut être reprise par une autre méthode de test structurel unitaire.

L'application du critère TLCM sur le graphe mixte d'une fonction sous test permet d'une part de limiter la combinatoire des chemins tout en garantissant le maintien de la couverture des chemins de la fonction sous test. De plus, l'application de ce critère permet la mise en place d'une couverture mixte de la fonction sous test dans le sens d'une couverture structurelle des chemins de la fonction sous test et d'une couverture fonctionnelle des domaines fonctionnels des fonctions imbriquées via une méthode de test en contexte.

L'application du critère TLCS sur le graphe mixte de la fonction sous test permet également la limitation de la combinatoire des chemins et le maintien de la couverture des chemins de la fonction sous test. Dans ce manuscrit, nous nous plaçons dans une optique de couverture de la fonction sous test et non dans une optique de test imbriqué. Lors de l'application du critère TLCS sur le graphe mixte, la redondance des tests des chemins structurels de la fonction sous test est éliminée.

Notre objectif est atteint par l'application des deux critères TLCM et TLCS : l'explosion combinatoire des chemins est limitée grâce à notre modélisation des chemins structurels contenant des instructions d'appel et la couverture de tous les k-chemins faisables de la fonction sous test est maintenue. L'utilisateur pourra donc choisir à sa guise et selon ses besoins et objectifs le critère de test à appliquer sur le graphe mixte de la fonction sous test.

.

Dans le chapitre suivant, ces deux stratégies vont être appliquées sur deux exemples de fonctions qui seront également soumises à deux traitements des appels de fonction plus classiques (un traitement "inlining" et l'utilisation de bouchons fonctionnels) pour les fonctions imbriquées.

Chapitre 13

Validation

Dans ce chapitre, nous allons soumettre deux exemples de fonctions sous test utilisant des fonctions imbriquées tout d'abord à un traitement "inlining" et à un traitement par bouchons fonctionnels. Nous verrons ainsi les limitations de ces deux techniques en terme d'explosion combinatoire des chemins et de non maintien de la couverture de 100% des chemins de la fonction sous test. Ces deux exemples de fonctions sous test utilisant des fonctions imbriquées seront ensuite modélisés sous forme de graphe mixte successivement soumis aux deux nouveaux critères de test définis.

Par les différents traitements de ces fonctions, nous pourrons ainsi illustrer plus en détails nos stratégies de gestion des appels de fonction et aussi leurs apports.

13.1 Analyse d'un premier exemple : appel de la fonction `maccarthy`

La fonction `maccarthy` est généralement utilisée en preuve de programme comme exemple afin de vérifier la correction de son algorithme et de prouver sa terminaison. Nous avons choisi d'utiliser cette fonction afin d'illustrer autre chose à savoir le gain obtenu en terme d'explosion combinatoire des chemins par l'application des critères TLCM et TLCS sur le graphe mixte de la fonction sous test.

13.1.1 Présentation de la fonction imbriquée `maccarthy`

Son implantation repose sur une récursivité double imbriquée qui introduit un grand nombre de chemins structurels (cf. figure 13.1). Elle est définie que sur un domaine en entrée entier et positif ou nul $[0, MaxInt]$. Elle retourne la valeur 91 pour toute valeur en entrée inférieure ou égale à 101 et elle retourne la valeur d'entrée soustraite de 10 pour toute valeur en entrée supérieure ou égale à 102 comme le définit sa spécification dans la figure 13.2.

Code source et spécification

Si nous déplions le code source de la fonction `maccarthy`, nous pouvons comptabiliser un nombre élevé de chemins correspondants aux cas suivants :

- un chemin structurel vérifiant la condition C `"(x>100)"` dans la structure conditionnelle (cf. figure 13.1 ce qui correspond au domaine d'entrée $[101, MaxInt]$
- 101 chemins structurels contenant un nombre différent d'instructions d'appel récursif, chacun de ses chemins correspondant à une des valeurs de $[0, 100]$.

```
1  int maccarthy(int x) {
2    if (x>100)
3      return (x-10);
4       else
5         return (maccarthy(maccarthy(x+11)));
6  }
7  int f(int x)
8  {
9    if ((x<0)) /*c1*/
10     x=-x;
11   x=maccarthy(x);
12   if (x<95)  /*c2*/
13     return(1);
14   else
15     return(0);
16
17 }
```

FIG. 13.1 – Implantations des fonctions `f` et `maccarthy`

```
1  FUNCTION maccarthy
2  /*@ requires
3  @ (w>=0)
4  @*/
5
6  /*@ ensures
7  @ ((w>101) => (z=w- 10))
8  @ ((w<=101)  => (z=91))
9  @*/
10
11 END
```

FIG. 13.2 – Spécification de la fonction `maccarthy`

En effet, si nous exécutons la fonction sur quelques valeurs en entrée du domaine $[0, 100]$, nous voyons en réalité que le double appel imbriqué correspond à incrémenter la valeur en entrée de la fonction `maccarthy` jusqu'à ce que celle-ci soit égale à 101.

Pour information, sur le domaine de définition de la fonction `maccarthy` $Def(maccarthy) = [0, MaxInt]$, il existe 102 chemins structurels exécutables au total.

13.1.2 Traitement "inlining" de la fonction `maccarthy`

Nous allons donc tester ici la fonction `f` appelant la fonction `maccarthy` comme présentée dans la figure 13.1 en utilisant un traitement "inlining" de la fonction imbriquée. Pour cela, nous soumettons la fonction `f` à la méthode de test PathCrawler initiale c'est-à-dire étendant le critère de test au code source de la fonction imbriquée `maccarthy`.

Remarque(s) 35
Nous avons dit dans le chapitre 7 que la méthode PathCrawler ne traite pas les fonctions récursives. En réalité, la méthode ne traite pas toutes les fonctions récursives. La fonction `maccarthy` *fait partie des exceptions car la difficulté du traitement des fonctions récursives concerne la possible perte de valeur des variables à cause d'un conflit de variables entre les différents appels de la même fonction. Le problème ne se pose pas pour la fonction* `maccarthy`.

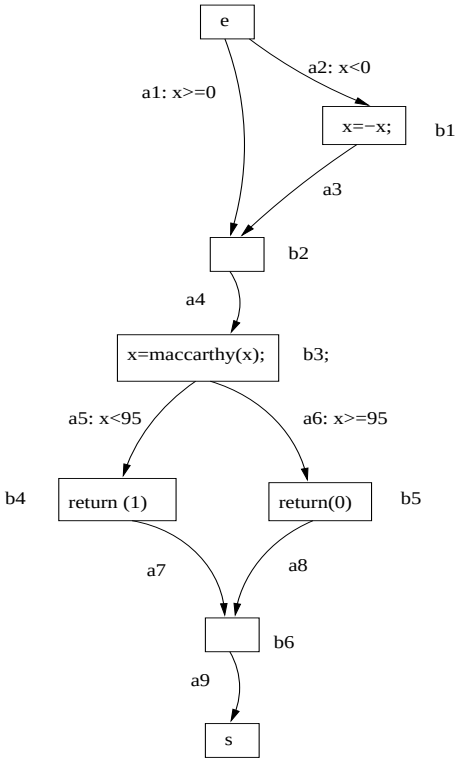

FIG. 13.3 – Graphe de contrôle de la fonction appelant la fonction `maccarthy`

La graphe de flot de contrôle de la fonction `f` est contenu dans la figure 13.3. Nous ne construisons pas celui de la fonction imbriquée `maccarthy` du fait que notre représentation des graphes de contrôle n'est pas adaptée aux fonctions récursives imbriquées sauf en cas de dépliage total de celui-ci ce qui induirait un graphe d'une taille trop conséquente.

La fonction `f` ne possède qu'une seule variable d'entrée de type entier telle que $p_1(X) = x$. La fonction `f` de profil

$$f : int \rightarrow int$$

ne possède pas de précondition donc le domaine fonctionnel de la fonction `f` est $Def(f) = [MinInt, MaxInt]$.

Application

Pour illustrer notre exemple, nous choisissons de limiter ce domaine de définition de la façon suivante : $Def_{|user}(f) = [-150, 150]$. Nous soumettons 3 fois successives la fonction `f` à la méthode PathCrawler avec un traitement "inlining" des fonctions imbriquées. Nous obtenons à chaque fois 205 cas de test. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 4.69, 4.63 et 3.26.

Analyse

Nous ne détaillons pas ici les différents cas de test obtenus en détail. Cependant nous avons vu dans la section précédente que 102 chemins structurels sont exécutables dans la fonction imbriquée `maccarthy` sur l'ensemble de son domaine de définition.
La première structure conditionnelle de la fonction sous test peut être caractérisée de la façon suivante :

- la première branche de la structure conditionnelle (arc $a2$ dans la figure 13.3) est activée pour toute valeur dans $[-150, -1]$ de la variable d'entrée de `f` et sa vérification est suivie de la soustraction de 0 par la valeur de la variable d'entrée de `f`.
- la seconde branche de la structure conditionnelle (arc $a1$ dans la figure 13.3) est activée pour toute valeur de $[0, 150]$ de la variable d'entrée de `f` et sa vérification n'introduit aucune définition de variables.

L'instruction d'appel de la fonction imbriquée `maccarthy` suit directement cette structure conditionnelle dans l'implantation de la fonction sous test. La fonction imbriquée est donc exécutée sur le domaine d'appel $[0, 150]$ si la branche de la structure conditionnelle correspondant à l'arc $a1$ est activée ce qui correspond à 102 chemins exécutables dans la fonction imbriquée `maccarthy`. De même, la fonction imbriquée est donc exécutée sur le domaine d'appel $[1, 150]$ si la branche de la structure conditionnelle correspondant à l'arc $a2$ est activée ce qui correspond à 101 chemins exécutables dans la fonction imbriquée.

L'instruction d'appel est suivie d'une seconde instruction conditionnelle qui teste la valeur de retour de la fonction imbriquée par rapport à la valeur 95 (arcs $a5$ et $a6$ dans la figure 13.3).

L'implantation de la fonction sous test correspond en tout à 4 chemins structurels exécutables. La fonction imbriquée quant à elle retourne soit la valeur 91 soit la valeur en entrée soustraite de 10. En raisonnant sur ces données, les chemins structurels de la fonction sous test `f` dépliés dans la fonction imbriquée `maccarthy` correspondent aux 8 cas suivants :

1. tout chemin déplié passant par les arcs $a2$ et $a5$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est 91[1],
2. tout chemin déplié passant par les arcs $a2$ et $a5$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est différente de 91[2],
3. tout chemin déplié passant par les arcs $a1$ et $a5$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est 91 ,
4. tout chemin déplié passant par les arcs $a1$ et $a5$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est différente de 91,
5. tout chemin déplié passant par les arcs $a2$ et $a6$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est 91,

[1] Nous rappelons que la fonction imbriquée `maccarthy` retourne 91 pour toute valeur en entrée dans $[0, 101]$
[2] Nous rappelons que la fonction imbriquée `maccarthy` retourne la valeur de la variable d'entrée soustraite de 10 pour toute valeur en entrée dans $[102, MaxInt]$

6. tout chemin déplié passant par les arcs $a2$ et $a6$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est différente de 91,
7. tout chemin déplié passant par les arcs $a1$ et $a6$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est 91.
8. tout chemin déplié passant par les arcs $a1$ et $a6$ du graphe de contrôle de la fonction sous test et dont la valeur de retour de la fonction imbriquée est différente de 91.

Reprenons ces différents cas par ordre pour raisonner en termes de sous-domaines d'entrée de la fonction sous test `f` et en nombre de chemins dépliés correspondant :

1. le sous-domaine en entrée de `f` associé est $[-101, -1]$ ce qui fait en tout 100 chemins dépliés,
2. le sous-domaine en entrée de `f` associé est $[-104, -102]$ ce qui fait 1 chemin déplié,
3. le sous-domaine en entrée de `f` associé est $[0, 101]$ ce qui fait en tout 101 chemins dépliés,
4. le sous-domaine en entrée de `f` associé est $[102, 104]$ ce qui fait 1 chemin déplié,
5. le sous-domaine en entrée de `f` associé est $\emptyset$ car si la fonction imbriquée retourne 91, la branche correspondante à l'arc $a6$ ne peut être activée,
6. le sous-domaine en entrée de `f` associé est $[-150, -105]$ ce qui fait 1 chemin déplié,
7. le sous-domaine en entrée de `f` associé est $\emptyset$ car si la fonction imbriquée retourne 91, la branche correspondante à l'arc $a6$ ne peut être activée et
8. le sous-domaine en entrée de `f` associé est $[105, 150]$ ce qui fait 1 chemin déplié.

Ce qui nous fait en tout 205 chemins dépliés exécutables que nous avons couverts en 205 cas de test.

Si nous regardons le graphe de flot de contrôle de la fonction sous test `f` dans la figure 13.3, nous observons que la fonction ne contient que 4 chemins structurels à couvrir. Un traitement "inlining" de la fonction imbriquée `maccarthy` nous a amené à couvrir plus de chemins que nécessaire : dans notre cas, 201 chemins supplémentaires ont été couverts.

13.1.3 Utilisation de bouchons fonctionnels

Si nous étudions la spécification de la fonction `maccarthy` de la figure 13.2, nous pouvons observer que celle-ci est composée de deux couples pre/post et donc de deux domaines fonctionnels :

$$Spec(maccarthy, W, Z) = \{PP_1(maccarthy, W, Z), PP_2(maccarthy, W, Z)\}$$

avec

$$PP_1(maccarthy, W, Z) = (Pre(maccarthy, W) \wedge D1(maccarthy, W), Q1(maccarthy, W, Z))$$

$$PP_2(maccarthy, W, Z) = (Pre(maccarthy, W) \wedge D2(maccarthy, W), Q2(maccarthy, W, Z))$$

et $p_1(W) = w$ et $p_1(Z) = z$

$$Pre(maccarthy, W) = w \geq 0$$

$$D1(maccarthy, W) = w > 101$$

$$D2(maccarthy, W) = w \leq 101$$

$$Q1(maccarthy, W, Z) = (z = w - 10)$$

$$Q2(maccarthy, W, Z) = (z = 91)$$

Les deux domaines fonctionnels sont donc

$$DF_1 = Dom(PP_1(maccarthy, W, Z))$$

$$DF_2 = Dom(PP_2(maccarthy, W, Z))$$

Pour l'utilisation de bouchons fonctionnels, le code source d'une fonction imbriquée n'est pas exploré. L'instruction d'appel est remplacée dans un bouchon fonctionnel qui retourne la même sortie que la fonction imbriquée qu'il remplace en se basant sur l'étude de sa spécification ainsi que la relation entrées/sorties vérifiées nécessaire pour le calcul du prédicat de chemin.

Pour illustrer l'utilisation d'un bouchon fonctionnel, nous construisons la représentation fonctionnelle de la fonction imbriquée sous forme de graphe abstrait ce qui signifie que les informations structurelles de la fonction imbriquée est abstraite par sa spécification. La fonction imbriquée est toujours exécutée dans le corps de la fonction sous test cependant lors du parcours du chemin d'exécution déplié suivi dans la fonction sous test et du calcul du prédicat de chemin associé, le chemin interne à la fonction imbriquée est abstrait par l'expression des contraintes du couple pre/post activé.

Nous modifions la stratégie d'exploration du graphe mixte de la fonction sous test de façon à ne jamais nier une contrainte interne à la fonction imbriquée ce qui correspond au cas de l'utilisation d'un bouchon fonctionnel.

Application

Nous appliquons donc la fonction sous test `f` sur le même domaine en entrée :
$Def_{|user}(f) = [-150, 150]$ et comme pour le traitement "inlining", nous répétons l'opération 3 fois.

Nous obtenons à chaque fois uniquement deux cas de test. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.00, 0.01 et 0.00.

Le premier cas de test est $X_1 = (-37)$. Le chemin couvert dans le graphe de flot de contrôle (cf. figure 13.3) est $Ch_1 = (e, a1, b2, a4, b3, a5, b4, a7, b6, a9, s)$. Le prédicat associé à ce chemin est $PC(Ch_1, X, f) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) \leq 101) \wedge (91 < 95)$.

Nous ne nions pas une contrainte interne à la fonction `maccarthy` ce qui signifie que le prochain cas de test correspond à $MaxC_1(X) = (p_1(X) \geq 0)$.

Le second cas de test est $X_2 = (57)$. Le chemin couvert dans le graphe de contrôle de la fonction sous test de la figure 13.3 est $Ch_2 = (e, a2, b1, a3, b2, a4, b3, a5, b4, a7, b6, a9, s)$.

Le prédicat de chemin associé est $PC(Ch_2, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) \leq 101) \wedge (91 < 95)$.

Toutes les contraintes à l'exception des contraintes imbriquées et des contraintes dont la négation correspond à un système insatisfiable ont été niées. Le test s'arrête.

L'arc $a6$ n'a jamais été couvert lors des cas de test car cette branche était inatteignable pour le domaine fonctionnel activé DF_1. Cependant, l'activation de l'autre domaine fonctionnel de la fonction imbriquée DF_2 aurait permis la couverture de cette branche.

Nous voyons ici que l'utilisation de bouchon fonctionnel pour le traitement des fonctions imbriquées ne permet pas de garantir le maintien de la couverture de la fonction sous test. Dans notre cas, nous n'avons couvert que 2 chemins de la fonction sous test alors que les 4 chemins de la fonction sous test sont faisables. Dans la figure 13.4, le chemin couvert lors du premier cas de test est mis en relief par des pointillés courts et le second chemin couvert par des pointillés longs.

Remarque(s) 36
Notons que le nombre de chemins couverts est aléatoire. Si le premier cas de test avait amené à couvrir le second domaine fonctionnel imbriqué correspondant à $(w \geq 0) \wedge (w > 101) => (z = w - 10)$) *dans la spécification (cf. figure 13.2), trois ou quatre des chemins de la fonction sous test auraient pu être couverts.*

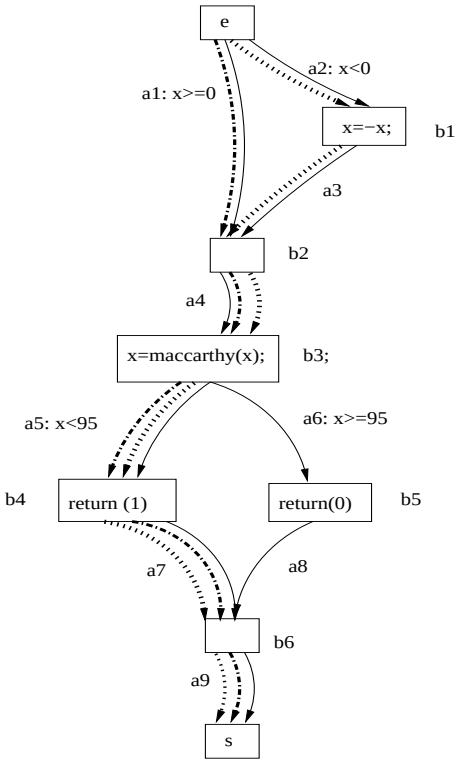

FIG. 13.4 – Chemins du CFG de la fonction sous test couverts lors de l'utilisation d'un bouchon fonctionnel

13.1.4 Application du critère TLCM au graphe mixte de la fonction sous test

Construction du graphe mixte

A partir du code source des fonctions, de l'interface et de la spécification (cf. figure 13.2) fournies par l'utilisateur, la représentation fonctionnelle de la fonction imbriquée ainsi que les contraintes associées aux différents domaines fonctionnels dans la table d'associations sont générés automatiquement.

D'après l'interface fournie par l'utilisateur, nous savons que la variable d'entrée fonctionnelle w correspond à la variable structurelle `x` qui est un paramètre passé par valeur dans la fonction `maccarthy` et que la variable fonctionnelle de sortie z correspond au retour de la fonction.

Nous pouvons construire le graphe mixte de la fonction sous test `f` qui va être soumis à la méthode PathCrawler selon le critère TLCM. La figure 13.5 contient le graphe mixte correspondant.

Application

Nous rappelons que la soumission du graphe mixte d'une fonction sous test selon le critère TLCM consiste à explorer en profondeur le graphe mixte de la fonction sous test et par conséquent à forcer le passage dans tous les domaines fonctionnels d'une fonction imbriquée pour un contexte d'appel donné.

Comme pour les cas précédents, nous appliquons 3 fois de suite la méthode. Nous obtenons à chaque fois 6 cas de test différents. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.01, 0.01 et 0.02.

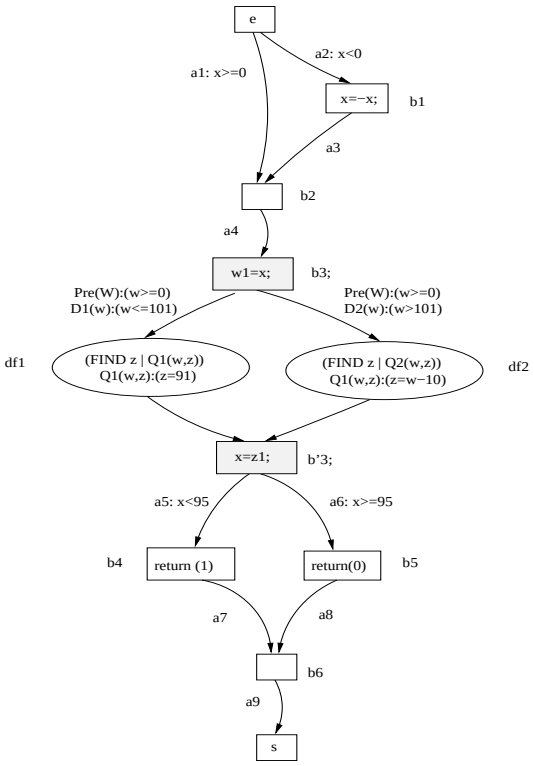

FIG. 13.5 – Graphe mixte de la fonction appelant la fonction `maccarthy`

La figure 13.6 contient la construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test `f`. Pour les contraintes de la fonction sous test, nous utilisons les annotations de son implantation contenues dans la figure 13.1. Les contraintes associées aux couples pre/post de la fonction imbriquée `maccarthy` sont notées $cdf1$ pour le premier domaine fonctionnel ($w \geq O \wedge w \leq 101$) et $cdf2$ pour le second domaine fonctionnel ($w \geq 0 \wedge w > 101$). L'arbre des chemins faisables couverts pour le i^{eme} cas de test est noté CTi.

Les chemins couverts dans le graphe mixte pour les différents cas de test sont illustrés dans la figure 13.7 à la page 197. Nous réutilisons les annotations de cette figure pour détailler les différents cas de test effectués.

Le premier cas de test est $X_1 = (38)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_1 = (e, a1, b2, a4, b3, a5', df1, a5", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT1 de la figure 13.7. Le premier domaine fonctionnel est donc couvert pour ce cas de test.
Le prédicat associé à ce chemin est $PC(Ch_1, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) \leq 101) \wedge (91 < 95)$ et $MaxC_1(X) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101)$.
Nous cherchons donc à couvrir le second domaine fonctionnel pour le même préfixe d'appel.

Le second cas de test est $X_2 = (118)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_2 = (e, a1, b2, a4, b3, a6', df2, a6", b3', a6, b5, a8, b6, a9, s)$ représenté par le graphe CT2 de la figure 13.7. Le second domaine fonctionnel est donc bien couvert ici.
Le prédicat associé à ce chemin est $PC(Ch_2, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101) \wedge (p_1(X) - 10 \geq 95)$ et $MaxC_2(X) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101) \wedge (p_1(X) - 10 < 95)$.
Le prochain cas de test doit couvrir le second suffixe en sortie du second domaine fonctionnel toujours pour le même préfixe d'appel.

Le troisième cas de test est $X_3 = (103)$. Le chemin couvert dans le graphe mixte de la fonction

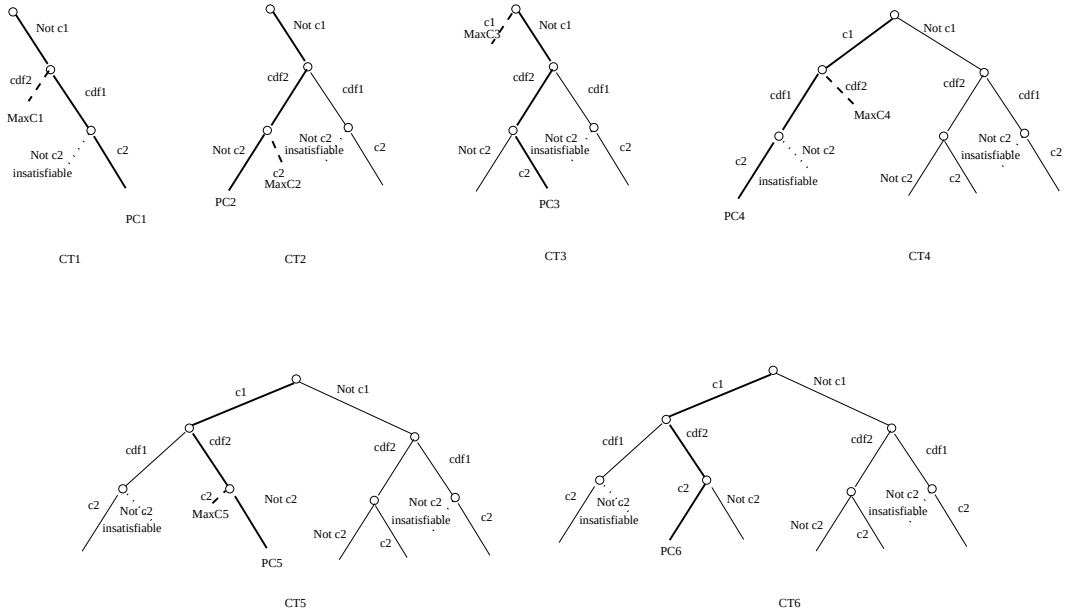

FIG. 13.6 – Construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test pour le critère TLCM

sous test est $Ch_3 = (e, a1, b2, a4, b3, a6', df2, a6", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT3 de la figure 13.7. Le second suffixe pour le second domaine fonctionnel est bien couvert par ce cas de test.
Le prédicat associé à ce chemin est $PC(Ch_3, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101) \wedge (p_1(X) - 10 < 95)$ et $MaxC_3(X) = (p_1(X) < 0)$.
Nous allons maintenant explorer un préfixe d'appel différent.

Le quatrième cas de test est $X_4 = (-76)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_4 = (e, a2, b1, a3, b2, a4, b3, a5', df1, a5", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT4 de la figure 13.7. Pour ce nouveau préfixe d'appel, le premier domaine fonctionnel est couvert.
Le prédicat associé à ce chemin est $PC(Ch_4, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) \leq 101) \wedge (91 < 95)$ et $MaxC_4(X) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101)$.
Nous désirons couvrir le second domaine fonctionnel pour ce nouveau préfixe d'appel .

Le cinquième cas de test est $X_1 = (-129)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_6 = (e, a2, b1, a3, b2, a4, b3, a6', df2, a6", b3', a6, b5, a8, b6, a9, s)$ représenté par le graphe CT5 de la figure 13.7. Le second domaine fonctionnel est couvert.
Le prédicat associé à ce chemin est $PC(Ch_5, X, f) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) > 101) \wedge (-p_1(X) - 10 \geq 95)$ et $MaxC_5(X) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) > 101) \wedge (-p_1(X) - 10 < 95)$.
Il nous reste donc à couvrir le second suffixe pour ce préfixe d'appel et le second domaine fonctionnel de la fonction imbriquée.

Le sixième et dernier cas de test est $X_1 = (-103)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_6 = (e, a2, b1, a3, b3, a6', df2, a6", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT6 de la figure 13.7. Le prédicat associé à ce chemin est $PC(Ch_6, X, f) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) > 101) \wedge (-p_1(X) - 10 < 95)$.

Si nous faisons la correspondance avec les chemins structurels du graphe de contrôle de la fonction sous test représentés dans la figure 13.3, les chemins couverts par ces cas de test correspondant aux séquences :

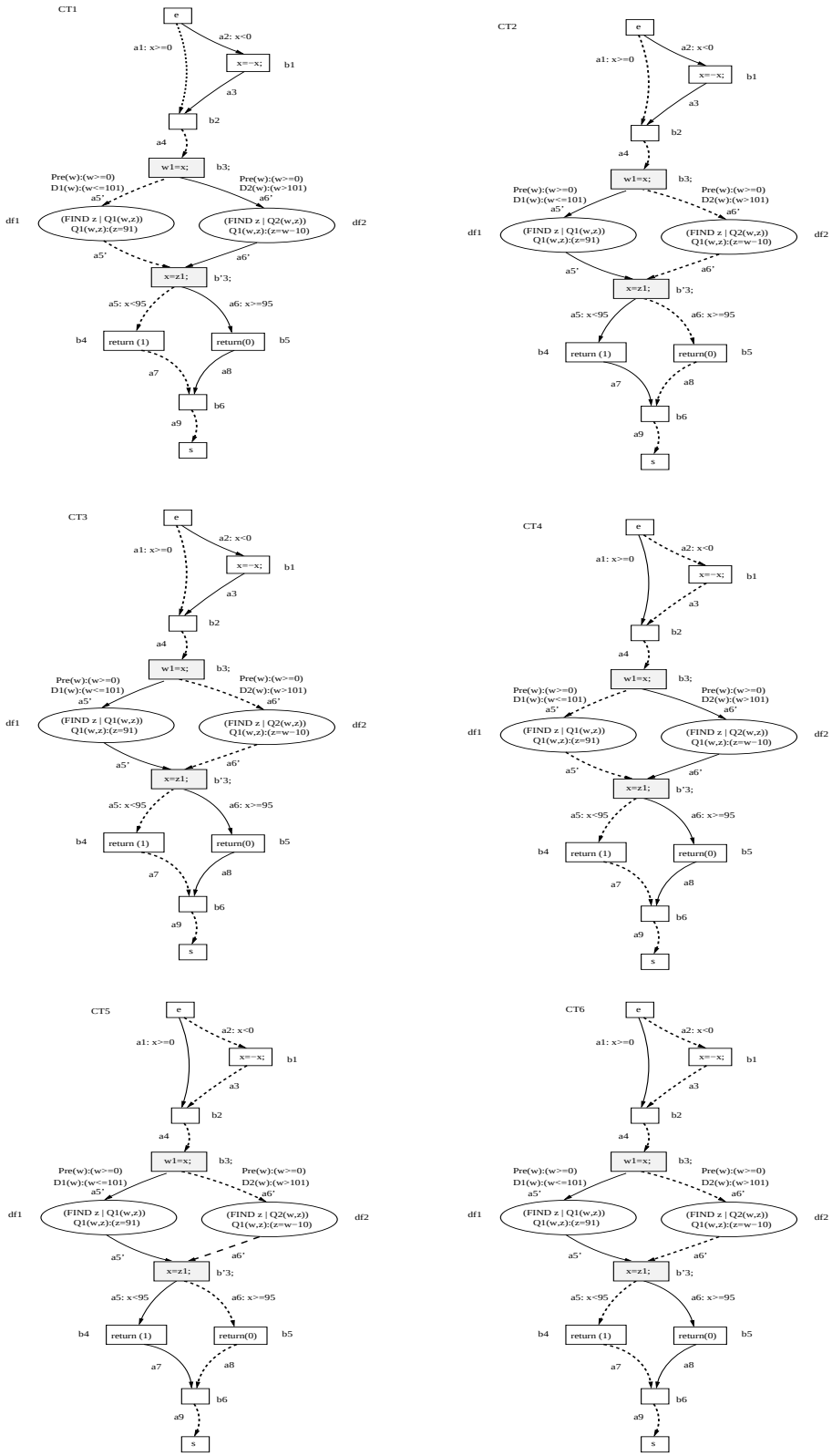

FIG. 13.7 – Chemins couverts dans le graphe mixte pour le critère TLCM

– $(e, a1, b2, a4, b3, a5, b4, a7, b6, a9, s)$ couvert deux fois (CT1 et CT3)
– $(e, a1, b2, a4, b3, a6, b5, a8, b6, a9, s)$ couvert une fois (CT2)
– $(e, a2, b1, a3, b2, a4, b3, a5, b4, a7, b6, a9, s)$ couvert deux fois (CT4 et CT6)
– $(e, a2, b1, a3, b2, a4, b3, a6, b5, a8, b6, a9, s)$ couvert une fois (CT5)

Nous gagnons en combinatoire des chemins par rapport à une méthode "inlining" (205 cas de test contre 6 cas de test). De plus, contrairement à l'utilisation de bouchons fonctionnels, nous maintenons le taux de couverture de 100% des chemins structurels de la fonction sous test.

Nous pouvons observer sur l'exemple précédent que 2 des chemins structurels de la fonction sous test sont couverts plusieurs fois c'est-à-dire que si nous raisonnons uniquement en terme de couverture des chemins structurels de la fonction sous test, il existe une redondance pour certains cas de test effectués (CT1 est équivalent à CT3 et CT4 est équivalent à CT6). Nous allons donc éliminer cette redondance en appliquant le critère TLCS sur le graphe mixte de la fonction sous test.

13.1.5 Application du critère TLCS au graphe mixte de la fonction sous test

Les trois soumissions successives de notre graphe mixte selon le critère TLCS nous amène à exécuter 4 cas de test différents. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.01, 0.01 et 0.01.

La figure 13.8 contient la construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test `f` selon l'application du critère TLCS. Nous rappelons que pour l'application du critère TLCS sur le graphe mixte de la fonction sous test, les domaines fonctionnels de la fonction imbriquée ne sont forcés que s'il existe des chemins en sortie de la fonction sous test encore non couverts. Nous réutilisons dans la figure 13.8 les mêmes annotations de contraintes que dans la figure 13.6 précédente. Le i^{eme} cas de test est noté CTi.

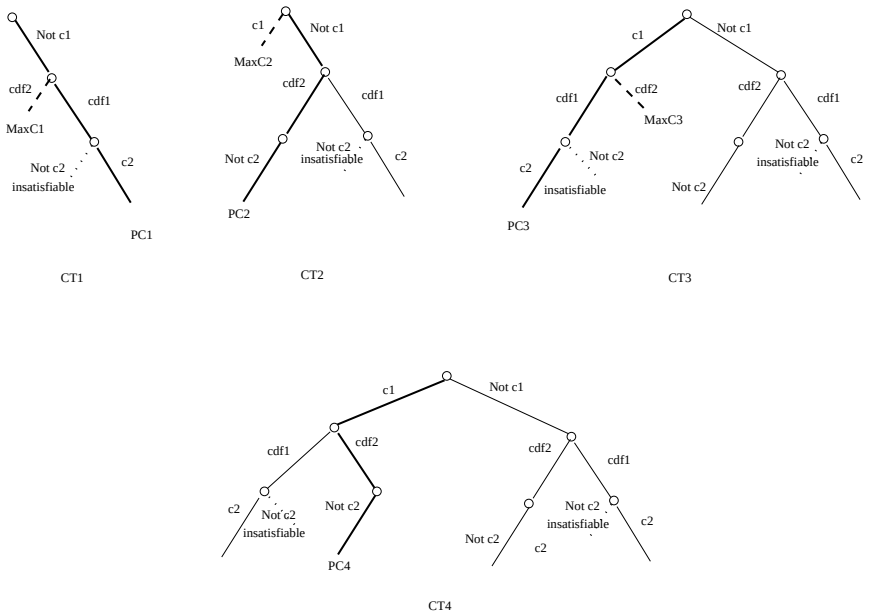

FIG. 13.8 – Construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test pour le critère TLCS

Le premier cas de test est $X_1 = (46)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_1 = (e, a1, b2, a4, b3, a5', df1, a5", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT1 de la figure 13.9. Le premier domaine fonctionnel est couvert. Le prédicat associé à ce chemin est $PC(Ch_1, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) \leq 101) \wedge (91 < 95)$ et $MaxC_1(X) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101) \wedge (p_1(X) - 10 > 95)$. Nous forçons le second domaine fonctionnel DF_2 pour ce préfixe d'appel car l'arc $a6$ n'a pu être couvert par le premier domaine fonctionnel DF_1 et nous forçons également le passage par cet arc $a6$ non couvert pour le prochain cas de test.

Le second cas de test est $X_2 = (131)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_2 = (e, a1, b2, a4, b3, a6', df2, a6", b3', a6, b5, a8, b6, a9, s)$ représenté par le graphe CT2 de la figure 13.9. Nous couvrons donc bien le second domaine fonctionnel et l'arc $a6$ que nous désirions atteindre. Le prédicat associé à ce chemin est $PC(Ch_2, X, f) = (p_1(X) \geq 0) \wedge (p_1(X) \geq 0) \wedge (p_1(X) > 101) \wedge (p_1(X) - 10 \geq 95)$ et $MaxC_2(X) = (p_1(X) < 0)$. Tous les chemins en sortie de la fonction imbriquée ont été couvert pour ce contexte d'appel donc la contrainte précédant l'appel imbriqué est niée c'est-à-dire que nous explorons un nouveau préfixe d'appel.

Le troisième cas de test est $X_3 = (-51)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_3 = (e, a2, b1, a3, b2, a4, b3, a5', df1, a5", b3', a5, b4, a7, b6, a9, s)$ représenté par le graphe CT3 de la figure 13.9. Le premier domaine fonctionnel est couvert pour ce nouveau préfixe d'appel.
Le prédicat associé à ce chemin est $PC(Ch_3, X, f) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) \leq 101) \wedge (91 < 95)$ et $MaxC_3(X) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) > 101) \wedge (-p_1(X) - 10 < 95)$. Comme pour le premier cas de test, un chemin en sortie de la fonction imbriquée n'a pas été couvert par le domaine fonctionnel activé DF_1, nous forçons donc la couverture de ce chemin par le second domaine fonctionnel imbriqué DF_2.

Le quatrième et dernier cas de test est $X_3 = (-130)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_4 = (e, a2, b1, a3, b2, a4, b3, a6', df2, a6", b3', a6, b5, a8, b6, a9, s)$ représenté par le graphe CT4 de la figure 13.9. Le chemin en sortie de la fonction imbriquée a bien été couvert en même temps que le second domaine fonctionnel pour ce préfixe d'appel.
Le prédicat associé à ce chemin est $PC(Ch_4, X, f) = (p_1(X) < 0) \wedge (-p_1(X) \geq 0) \wedge (-p_1(X) > 101) \wedge (-p_1(X) - 10 \geq 95)$.

Les chemins couverts dans le graphe mixte pour le critère TLCS sont illustrés par la figure 13.9.

Si nous faisons la correspondance avec les chemins structurels du graphe de contrôle de la fonction sous test représenté dans la figure 13.3, les chemins couverts par ces cas de test sont les séquences :

- $(e, a1, b2, a4, b3, a5, b4, a7, b6, a9, s)$ (CT1)
- $(e, a1, b2, a4, b3, a6, b5, a8, b6, a9, s)$ (CT2)
- $(e, a2, b1, a3, b2, a4, b3, a5, b4, a7, b6, a9, s)$ (CT3)
- $(e, a2, b1, a3, b2, a4, b3, a6, b5, a8, b6, a9, s)$ (CT5)

Le tableau 13.1 contient la synthèse des résultats obtenus sur ce premier exemple. Avec l'application du critère TLCS, nous gagnons en combinatoire des chemins par rapport à une méthode "inlining" (205 cas de test) mais aussi par rapport à l'application du critère TLCM sur le graphe mixte de la fonction sous test. En effet, nous ne couvrons qu'une seule fois chaque chemin du graphe de contrôle de la fonction sous test, nous avons donc supprimé la redondance des cas de test structurels de la fonction sous test.

De plus, comme l'application du critère TLCM, nous maintenons le taux de couverture de 100% des chemins structurels de la fonction sous test : les 4 chemins structurels de `f` ont été couverts en 4 cas de test.

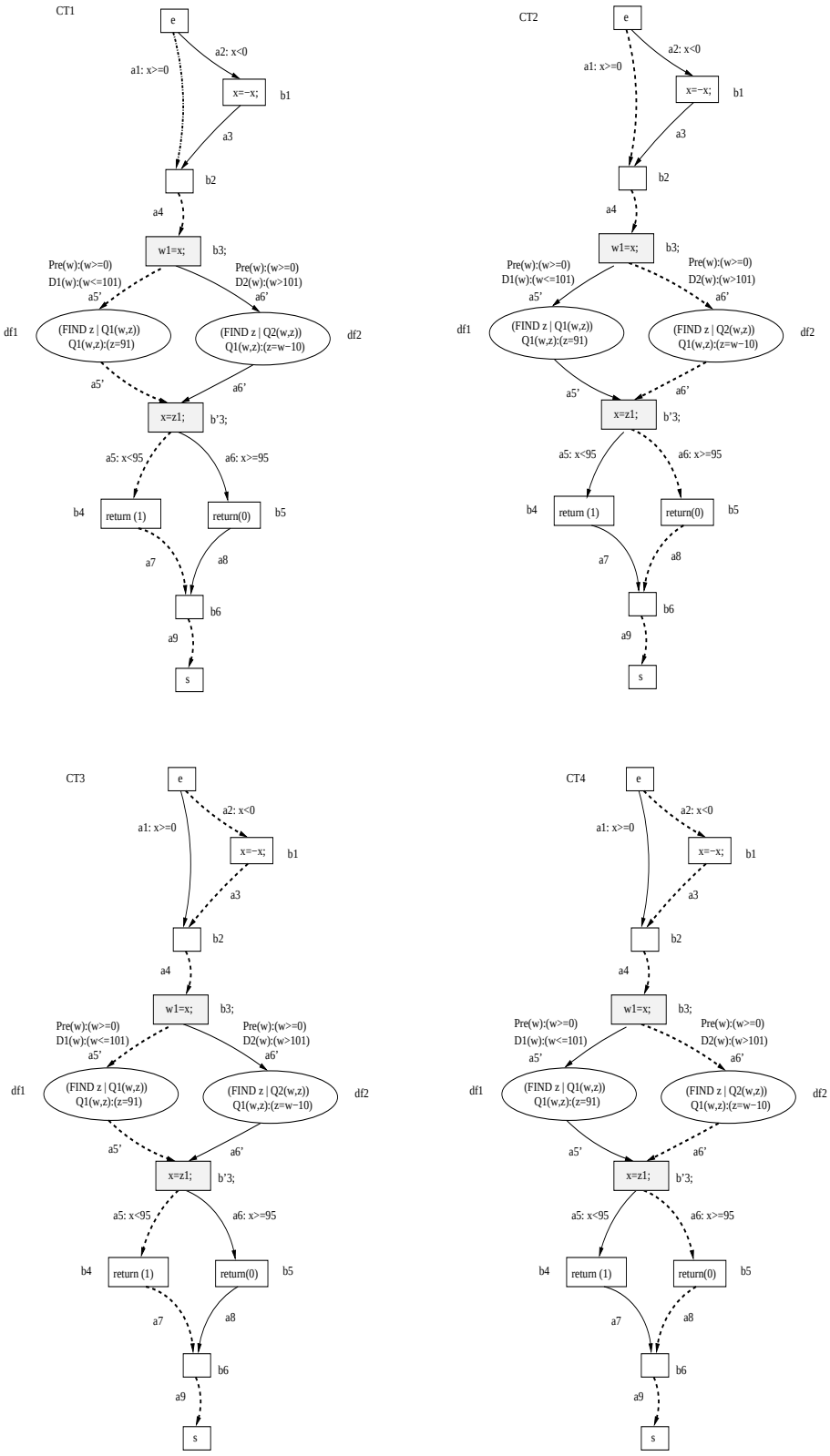

FIG. 13.9 – Chemins couverts dans le graphe mixte pour le critère TLCS

	"inlining"	bouchon fonctionnel	TLCM[a]	TLCS[b]
nombre de cas de test	205	2	6	4
% des chemins couverts de `f`	100	50	100	100
temps d'exécution[c]	4.69 4.63 3.26	0.00 0.01 0.00	0.01 0.01 0.02	0.01 0.01 0.01
cas de test redondants pour `f`	201	0	2	0

[a] critère tous-les-chemins-mixtes
[b] critère tous-les-chemins-structurels
[c] temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux

TAB. 13.1 – Différents résultats obtenus pour le test de la fonction appelant la fonction `maccarthy`

13.2 Second exemple : appel de la fonction `delta_tab`

13.2.1 Présentation de la fonction imbriquée `delta_tab`

La fonction `delta_tab` détermine dans quel intervalle des éléments d'un tableau constant de taille fixe se trouve une valeur passée en paramètre de la fonction. Il s'agit d'une fonction régulièrement utilisée en traitement du signal par exemple : le tableau constant contenant les valeurs correspondant à une discrétisation des valeurs d'une fonction correspondant à une linéarisation de fonction par morceaux.

13.2.2 Code source et spécification

Si nous analysons le code source de la figure 13.10 de la fonction `delta_tab` et son graphe de contrôle présenté dans la figure 13.12, nous pouvons comptabiliser en tout 13 chemins exécutables correspondant à :

- la premier structure conditionnelle de la figure 13.10 est vérifiée ce qui correspond au domaine structurel d'entrée $[MinInt, -61]$,
- la seconde structure conditionnelle de la figure 13.10 est vérifiée ce qui correspond au domaine structurel d'entrée $[60, MaxInt]$,
- les deux structures conditionnelles ne sont pas vérifiées et la structure répétitive est exécutée 11 fois successives ce qui correspond au domaine structurel d'entrée $[-60, 59]$. Pour l'ensemble des itérations de la structure répétitive, la structure conditionnelle imbriquée est vérifiée une unique fois : chaque vérification de la structure conditionnelle à un nombre d'itérations différent introduit un nouveau chemin d'exécution. Pour ce dernier scénario, nous avons donc 11 chemins d'exécution distincts.

La spécification de la fonction de la figure 13.11 définit trois domaines fonctionnels tels que $[MinInt, -61]$, $[60, MaxInt]$, $[-60, 59]$.

13.2.3 Traitement "inlining" de la fonction `delta_tab`

Nous allons donc tester ici la fonction `f` appelant la fonction `delta_tab` comme présentée dans la figure 13.10 en utilisant un traitement "inlining" de la fonction imbriquée. Pour cela, nous soumettons la fonction `f` à la méthode de test PathCrawler initiale c'est-à-dire étendant le critère de test au code source de la fonction imbriquée `delta_tab`.

La graphe de flot de contrôle de la fonction `f` est contenu dans la figure 13.13 et celui de la fonction imbriquée `delta_tab` est contenue dans la figure 13.12.

La fonction `f` possède trois variables d'entrée dont deux variables d'entrée constantes correspondant aux variables globales `t` et `n`. La fonction `f` de profil ne possède pas de précondition et seule la valeur de sa troisième entrée (la variable `valeur`) peut changer d'une exécution à l'autre donc nous ne considérerons que le domaine de cette variable en entrée soit $[MinInt, MaxInt]$.

```
1  const int n=12;
2
3  const int t[12]={-60,-55,-40,-25,-15,-5,5,15,25,40,55,60};
4
5  int delta_tab(int valeur)
6  {
7    int r;
8    int i;
9    if (valeur < t[0])
10     r=-2;
11   else
12     if (valeur>=t[n-1])
13       r=-1;
14    else
15       for(i=0;i<n-1;i++)
16         {
17           if ((valeur>=t[i])&&(valeur<t[i+1]))
18             r=i;
19         }
20   return (r);
21 }
22
23 int f(int valeur)
24 {
25   int i;
26   int retour;
27
28   i=delta_tab(valeur);
29   if ((i==-1)||(i==-2))
30     retour=0;
31   else
32     if ((i>=0)&&(i<n-1))
33       retour=1;
34
35     else
36       retour=-1;
37   return(retour);
38 }
```

FIG. 13.10 – Code source de la fonction sous test et de la fonction imbriquée `delta_tab`

```
1  FUNCTION delta_tab
2  /*@ requires
3  @ (true)
4  @*/
5
6  /*@ ensures
7  @ ((w1[0]>w3) => (z1=-2))
8  @ ((w1[w2- 1]<=w3)  => (z1=-1))
9  @ ((w1[0]<=w3)&&(w1[w2- 1]>w3)  => (exist i ->
10 (i>=0)&&(i<w2-1)&&(w3>=w1[i])&&(w3<=w1[i+1]) &&(z1=i))
11 @*/
12 END
```

FIG. 13.11 – Spécification de la fonction `delta_tab`

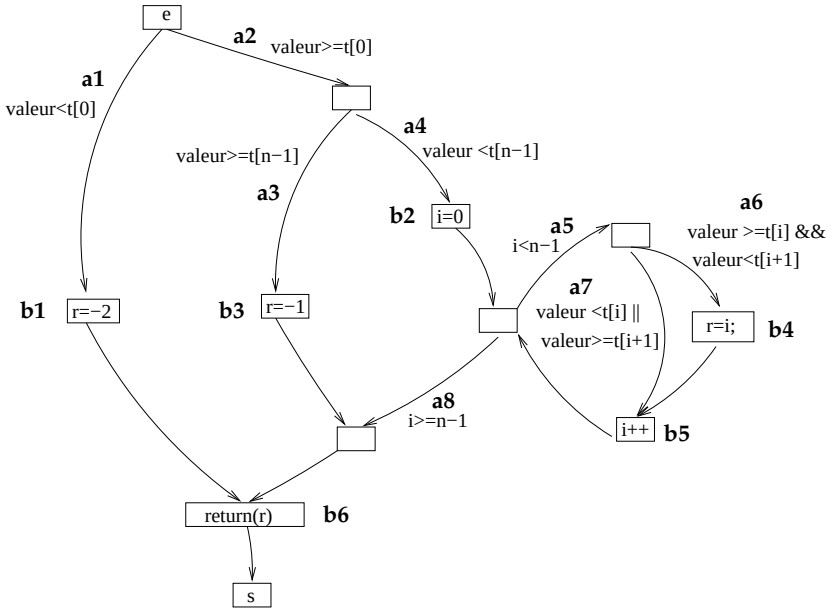

FIG. 13.12 – Graphe de flot de contrôle de la figure `delta_tab`

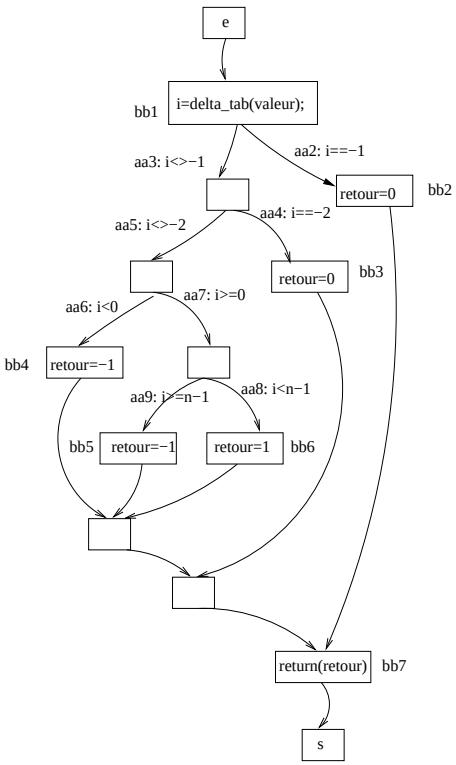

FIG. 13.13 – Graphe de contrôle de la fonction appelant la fonction `delta_tab`

Application

Pour illustrer notre exemple, nous choisissons de limiter le domaine de définition de la fonction sous test en limitant la variable `valeur` à l'intervalle $[-100, 100]$. Nous soumettons 3 fois successives

la fonction `f` à la méthode PathCrawler avec un traitement "inlining" des fonctions imbriquées. Nous obtenons à chaque fois 13 cas de test. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.03, 0.05 et 0.05.

Analyse

Nous ne détaillons pas ici les différents cas de test obtenus en détails. Cependant nous avons vu dans la section précédente que 13 chemins structurels sont exécutables dans la fonction imbriquée `delta_tab` sur l'ensemble de son domaine de définition et la fonction appelante possède, quant à elle, 3 chemins structurels faisables. Sur ces trois chemins structurels faisables, un seul chemin est exécutable pour chaque domaine fonctionnel exercé dans la fonction imbriquée. Cela signifie que le graphe déplié contient exactement 13 chemins exécutables structurels.

L'instruction d'appel de la fonction imbriquée `delta_tab` correspond à un bloc obligatoire de la fonction sous test c'est-à-dire que toute exécution de la fonction sous test entraîne obligatoirement l'exécution de la fonction imbriquée. Nous pouvons voir dans la figure 13.13 que le bloc d'appel suit directement le bloc d'entrée de la fonction et ce, sans aucune condition.

Le traitement "inlining" de la fonction sous test revient à couvrir tous les chemins faisables du graphe déplié de la fonction `f` présenté dans la figure 13.14. Nous réutilisations les notations de cette figure pour décrire un exemple de 13 cas de test obtenus :

- $valeur = 15$: la troisième structure conditionnelle de la fonction imbriquée est vérifiée (passage, entre autres, par $a2, a4, a6$) et celle-ci retourne l'indice 7 (passage par $a7$ à la huitième itération de la structure répétitive),le chemin suivi dans la fonction sous test est $(e, bb1, aa3, aa5, aa7, aa8, bb6, s)$,
- $valeur = 39$: idem mais la fonction imbriquée retourne l'indice 8 (passage par $a7$ à la neuvième itération de la structure répétitive), le chemin suivi dans la fonction sous test est le même,
- $valeur = 46$: idem mais la fonction imbriquée retourne l'indice 9 (passage par $a7$ à la dixième itération de la structure répétitive),
- $valeur = 56$: idem mais la fonction imbriquée retourne l'indice 10 (passage par $a7$ à la onzième itération de la structure répétitive),
- $valeur = 11$: idem mais la fonction imbriquée retourne l'indice 6 (passage par $a7$ à la septième itération de la structure répétitive),
- $valeur = -1$: idem mais la fonction imbriquée retourne l'indice 5 (passage par $a7$ à la sixième itération de la structure répétitive),
- $valeur = -10$: idem mais la fonction imbriquée retourne l'indice 4 (passage par $a7$ à la cinquième itération de la structure répétitive),
- $valeur = -19$: idem mais la fonction imbriquée retourne l'indice 3 (passage par $a7$ à la quatrième itération de la structure répétitive),
- $valeur = -31$: idem mais la fonction imbriquée retourne l'indice 2 (passage par $a7$ à la troisième itération de la structure répétitive),
- $valeur = -48$: idem mais la fonction imbriquée retourne l'indice 1 (passage par $a7$ à la deuxième itération de la structure répétitive),
- $valeur = -59$:idem mais la fonction imbriquée retourne l'indice 0 (passage par $a7$ à la première itération de la structure répétitive),
- $valeur = 81$: la seconde structure conditionnelle de la fonction imbriquée est vérifiée (chemin imbriqué : $(e, a2, a3, b3, s)$) et celle-ci retourne -1, le chemin suivi dans la fonction sous test est $(e, bb1, aa2, s)$,
- $valeur = -74$: la première structure conditionnelle de la fonction imbriquée est vérifiée (chemin imbriqué : $(e, a1, b1, s)$) et celle-ci retourne -2, le chemin suivi dans la fonction sous test est $(e, bb1, aa3, aa4, bb3, s)$.

Nous avons bien couvert les 13 chemins exécutables du graphe déplié de la fonction sous test (cf. figure 13.14).

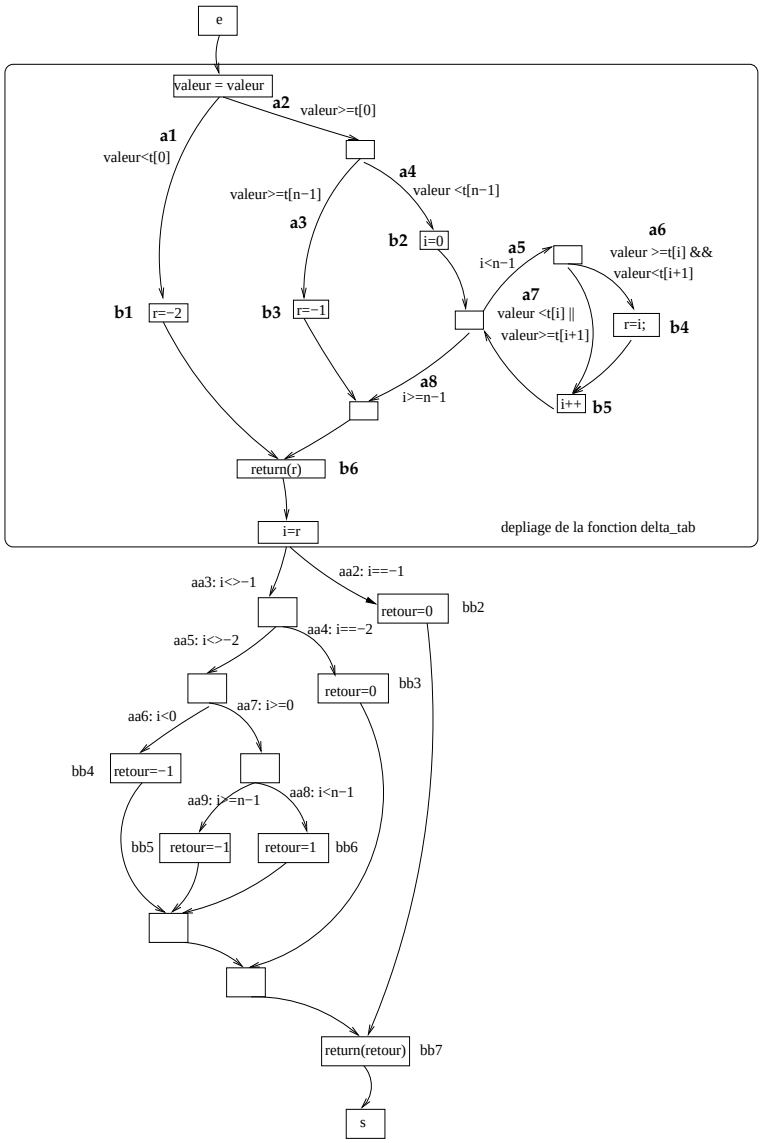

FIG. 13.14 – Graphe de contrôle de la fonction `f` avec dépliage du graphe de la fonction imbriquée `delta_tab`

Si nous raisonnons uniquement sur les chemins faisables du graphe de contrôle la fonction sous test sans dépliage de la figure 13.13, nous n'avions que 3 chemins à couvrir donc la méthode "inlining" nous a amené a exécuter 10 cas de test inutiles. En effet, si nous reprenons les cas de test présentés, nous pouvons observer que les 11 premiers cas de test reviennent à couvrir un seul et même chemin structurel de la fonction sous test.

13.2.4 Utilisation de bouchons fonctionnels

Si nous étudions la spécification de la fonction `delta_tab` de la figure 13.11, nous pouvons observer que celle-ci est composée de trois couples pre/post et donc de trois domaines fonctionnels :

$$Spec(delta\_tab, W, Z) = \{PP_1(delta\_tab, W, Z), PP_2(delta\_tab, W, Z), PP_3(delta\_tab, W, Z)\}$$

avec

$$PP_1(delta\_tab, W, Z) = (Pre(delta\_tab, W) \wedge D1(delta\_tab, W), Q1(delta\_tab, W, Z))$$

$$PP_2(delta\_tab, W, Z) = (Pre(delta\_tab, W) \wedge D2(delta\_tab, W), Q2(delta\_tab, W, Z))$$

$$PP_3(delta\_tab, W, Z) = (Pre(delta\_tab, W) \wedge D3(delta\_tab, W), Q3(delta\_tab, W, Z))$$

et $p_1(W) = w1, p_2(W) = w2, p_3(W) = w3$ et $p_1(Z) = z1$ avec $w1$ le tableau constant, $w2$ la taille constante de ce tableau et $w3$ la valeur recherchée

$$Pre(delta\_tab, W) = true$$

$$D1(delta\_tab, W) = w3 < w1[0]$$

$$D2(delta\_tab, W) = w3 \geq w1[0] \wedge w3 \geq w1[w2 - 1]$$

$$D3(delta\_tab, W) = w3 \geq w1[0] \wedge w3 < w1[w2 - 1] \wedge w3 \geq w1[i] \wedge w3 < w1[i + 1]$$

$$Q1(delta\_tab, W, Z) = (z1 = -2)$$

$$Q2(delta\_tab, W, Z) = (z1 = -1)$$

$$Q3(delta\_tab, W, Z) = (z1 \leq 0 \wedge z1 < w2 - 1 \wedge w3 < w1[z1 + 1] \wedge w3 \geq w1[z1])$$

Les trois domaines fonctionnels sont donc

$$DF_1 = Dom(PP_1(delta\_tab, W, Z))$$

$$DF_2 = Dom(PP_2(delta\_tab, W, Z))$$

$$DF_3 = Dom(PP_3(delta\_tab, W, Z))$$

Pour l'utilisation de bouchons fonctionnels, le code source d'une fonction imbriquée n'est pas exploré. les informations internes à la fonction imbriquée sont abstraites par les informations fonctionnelles issues de sa spécification comme pour le précédent exemple.

Lors du parcours du chemin déplié dans la fonction sous test et du calcul du prédicat de chemin associé, le chemin interne à la fonction imbriquée est abstrait par les contraintes associées du couple pre/post activé pour le cas de test courant.

Pour cette illustration, nous modifions la stratégie de façon à ne jamais nier une contrainte interne à la fonction imbriquée ce qui correspond au cas de l'utilisation de bouchon fonctionnel.

Application

Nous appliquons donc la fonction sous test `f` sur le même domaine en entrée et comme pour le traitement "inlining", nous répétons l'opération 3 fois.

Nous obtenons à chaque fois un unique cas de test. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.00, 0.00 et 0.01.

Un exemple de cas de test obtenu est $X_1 = (24)$. Le chemin couvert dans le graphe de flot de contrôle (cf. figure 13.13) est $Ch_1 = (e, bb1, aa3, aa5, aa7, aa8, bb6, s)$. Le prédicat associé à ce chemin ne s'obtient pas directement ici car il faut identifier la relation entre la valeur en entrée et en sortie de la fonction imbriquée est $PC(Ch_1, X, f) = (p_1(X)[0] =< p_3(X)) \wedge (p_1(X)[p_2(X) - 1] > p_3(X)) \wedge (p_3(X) >= p_2(X)[7]) \wedge (p_1(X) =< t[8]) \wedge (7 \neq -1) \wedge (7 \neq -2) \wedge (7 \geq 0) \wedge (7 < p_3(X) - 1)$ avec $p_3(X)$ la valeur entière passée en paramètre de `f`, $p_1(X)$ la tableau constant `t` et $p_2(X)$ la taille constante `n` de ce tableau et 7 la valeur retournée par notre bouchon fonctionnel correspondant à l'indice de borne droite de l'intervalle du tableau contenant la valeur recherchée.

La négation des deux dernières contraintes du prédicat de chemin aboutissent à des systèmes insatisfiables. Les autres contraintes du prédicat étant internes à la fonction imbriquée, elles ne sont pas niées comme pour tout bouchon fonctionnel. Aucun backtrack n'est possible en amont de

l'appel de la fonction `delta_tab` puisqu'aucune contrainte interne à la fonction sous test précède l'instruction d'appel.

Toutes les contraintes à l'exception des contraintes imbriquées et des contraintes dont la négation correspond à un système insatisfiable ont été niées. Le test s'arrête.

Un seul chemin de la fonction sous test `f` a été couvert alors que nous savons qu'il en existe trois de faisables. Nous rappelons que chacun de ces chemins est faisable pour un domaine fonctionnel donné de la fonction imbriquée. Nous avons, dans ce cas, activé uniquement le domaine fonctionnel DF_3 alors que l'activation des autres domaines fonctionnels de la fonction imbriquée (DF_1 et DF_2) aurait permis la couverture des deux autres chemins exécutables de la fonction sous test.

Nous voyons ici que l'utilisation de bouchon fonctionnel pour le traitement des fonctions imbriquées ne permet pas de garantir le maintien de la couverture de la fonction sous test. Dans notre cas, nous n'avons couvert que 1 seul chemin de la fonction sous test soit seulement $1/3$ des chemins faisables. Dans la figure 13.15, le chemin couvert lors de cet unique cas de test est mis en relief par des pointillés.

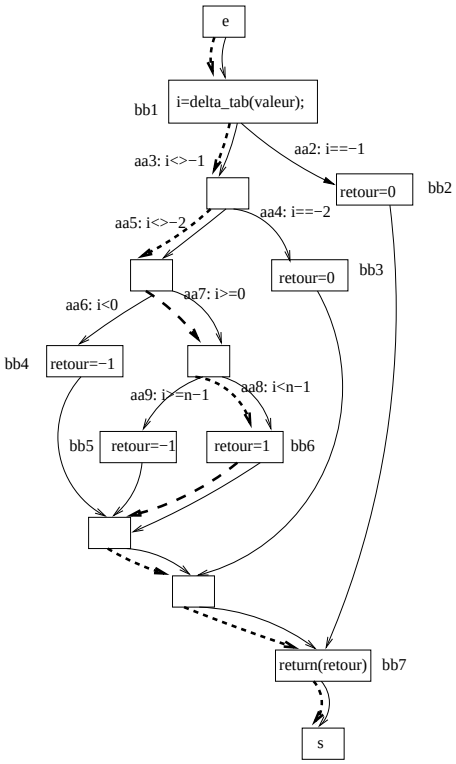

FIG. 13.15 – Chemins du CFG de la fonction sous test couverts lors de l'utilisation d'un bouchon fonctionnel

Remarque(s) 37
Notons que le nombre de chemins couverts n'est pas aléatoire pour cet exemple. Chaque exécution de la fonction sous test selon un traitement "inlining" de la fonction imbriquée ne permet de couvrir qu'un seul et unique chemin à la fois.

13.2.5 Graphe mixte de la fonction sous test et critère TLCM

Construction du graphe mixte

A partir du code source des fonctions, de l'interface et de la spécification (cf. figure 13.11) fournies par l'utilisateur, la représentation fonctionnelle de la fonction imbriquée ainsi que les contraintes associées aux différents domaines fonctionnels dans la table d'associations sont générés automatiquement.

L'interface nous indique que la variable d'entrée fonctionnelle $w1$ correspond à la variable structurelle globale `t`, que la variable fonction $x2$ correspond à la variable structurelle globale `n` et que la variable d'entrée fonctionnelle $w3$ correspond à la variable structurelle `valeur` passée en paramètre de la fonction `delta_tab`. Pour les sorties, nous en déduisons que la variable fonctionnelle de sortie $z1$ correspond au retour de la fonction.

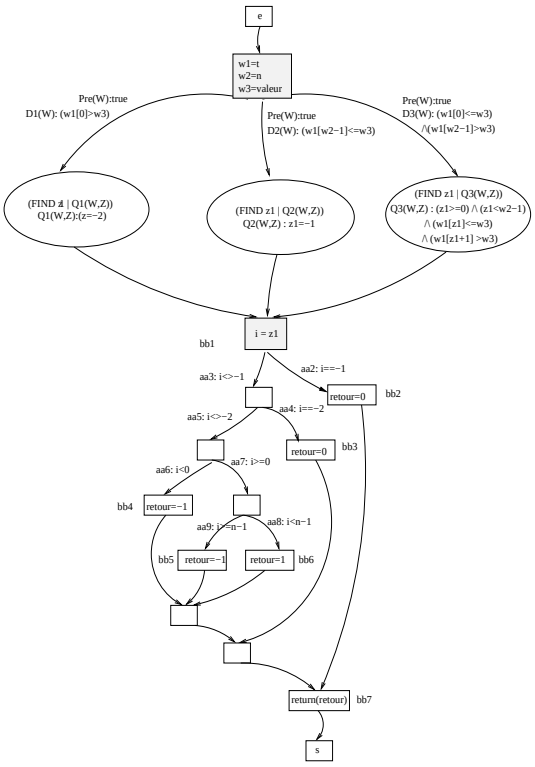

FIG. 13.16 – Graphe mixte de la fonction appelant la fonction `delta_tab`

Nous pouvons construire le graphe mixte de la fonction sous test `f` tel qu'il est présenté dans la figure 13.16 pour le soumettre à la méthode PathCrawler

Application

La soumission du graphe mixte d'une fonction sous test selon le critère TLCM consiste à explorer en profondeur le graphe mixte de la fonction sous test en profondeur d'abord et donc à couvrir tous les chemins structurels faisables de la fonction sous test et tous les domaines fonctionnels de la fonction imbriquée par contexte d'appel. Pour la fonction qui nous intéresse, il n'existe qu'un seul chemin partiel amenant à l'instruction d'appel et donc un unique contexte d'appel.

Comme pour les cas précédents, nous appliquons 3 fois de suite la méthode. Nous obtenons à chaque fois 3 cas de test différents. Les temps d'exécution CPU en secondes sur un PC de 2GHZ

fonctionnant sous Linux sont : 0.02, 0.03 et 0.02.

La figure 13.17 contient la construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test `f`. Pour les contraintes de la fonction sous test, nous utilisons les annotations de son implantation contenues dans la figure 13.10. Les contraintes associées aux couples pre/post de la fonction imbriquée `delta` sont notées $cdfi$ pour le i^{eme} domaine fonctionnel. Le i^{eme} cas de test est noté CTi.

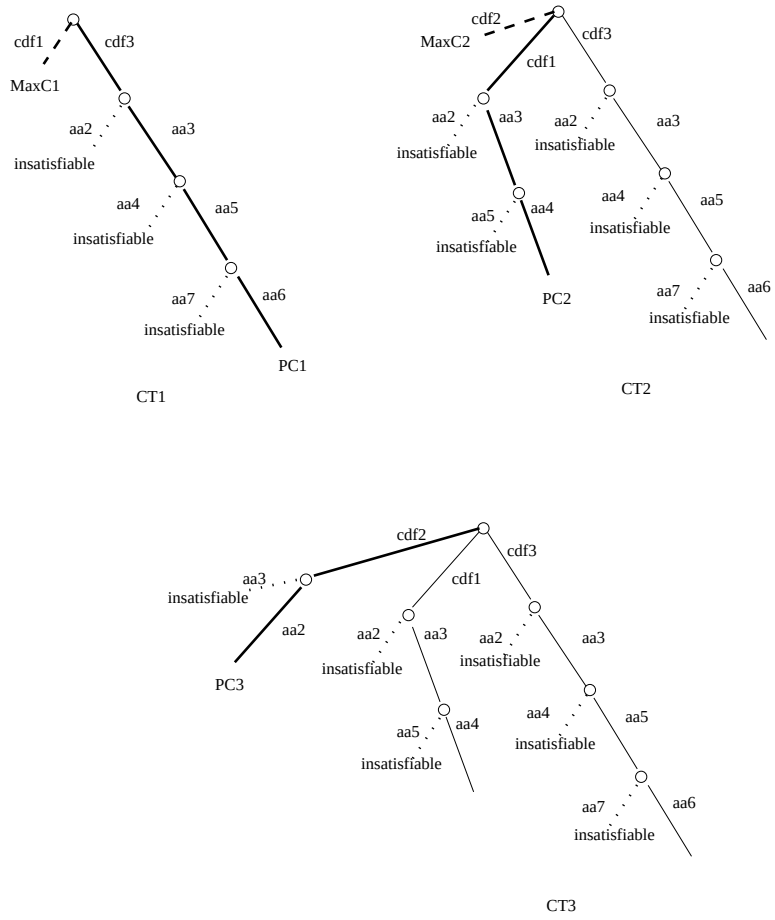

FIG. 13.17 – Construction de l'arbre des chemins faisables du graphe mixte de la fonction sous test selon le critère TLCM

Le premier cas de test est $X_1 = (8)$[3]. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_1 = (e, cdf3, aa3, aa5, aa7, aa8, bb6, s)$ représenté par le graphe CT1 de la figure 13.18. Le troisième domaine fonctionnel est donc couvert pour ce cas de test.
Le prédicat associé à ce chemin est $PC(Ch_1, X, f) = (p_1(X)[0] \leq p_3(X)) \wedge (p_1(X)[p_2(X) - 1] > p_3(X)) \wedge (6 \neq -1) \wedge (6 \neq -2) \wedge (6 \geq 0) \wedge (6 < p_2(X) - 1)$ avec 6 la valeur retournée par la fonction `delta_tab` ce qui signifie que la valeur 8 est contenue dans l'intervalle de valeurs $[t[6], t[7]]$. Un seul chemin de la fonction sous test étant exécutable pour un domaine fonctionnel imbriqué donné, nous couvrons donc un autre domaine fonctionnel à savoir la premier, nous avons donc $MaxC_1(X) = (p_1(X)[0] < p_3(X))$.

[3] Les variables d'entrée de la fonction étant des constantes à l'exception du paramètre `valeur`, nous ne précisons que la valeur de celle-ci pour chaque cas de test

Le second cas de test est $X_2 = (-74)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_2 = (e, cdf1, aa3, aa4, bb3, s)$ représenté par le graphe CT2 de la figure 13.18. Le premier domaine fonctionnel est donc bien couvert ici.
Le prédicat associé à ce chemin est $PC(Ch_2, X, f) = (p_1(X)[0] > p_3(X)) \wedge (-2 \neq -1) \wedge (-2 = -2)$. Nous forçons alors le dernier domaine fonctionnel à savoir le second $MaxC_2(X) = (p_1(X)[p_2(X) - 1] \leq p_3(X))$.

Le troisième cas de test est $X_3 = (80)$. Le chemin couvert dans le graphe mixte de la fonction sous test est $Ch_3 = (e, cdf2, aa2, bb2, s)$ représenté par le graphe CT3 de la figure 13.18. Le troisième et dernier chemin partiel faisable en sortie de la fonction imbriquée est donc bien couvert par ce cas de test.
Le prédicat associé à ce chemin est $PC(Ch_3, X, f) = (p_1(X)[p_2(X) - 1] \leq p_3(X)) \wedge (-1 = -1)$.
Tous les domaines fonctionnels ont été forcés, tous les chemins partiels en sortie faisables ont été couverts, le test est donc terminé.

Les chemins couverts dans le graphe mixte sont illustrés par la figure 13.18.

Si nous faisons la correspondance avec les chemins structurels du graphe de contrôle de la fonction sous test représenté dans la figure 13.13, les chemins couverts par ces cas de test sont les séquences :

- $(e, bb1, aa3, aa5, aa7, aa8, bb6, s)$ couvert une fois (CT1)
- $(e, bb1, aa3, aa4, bb3, s)$ couvert une fois (CT2)
- $(e, bb1, aa2, bb2, s)$ couvert une fois (CT3)

Nous gagnons en combinatoire des chemins par rapport à une méthode "inlining" (13 cas de test contre 3 cas de test). Contrairement à l'utilisation de bouchons fonctionnels, nous maintenons le taux de couverture de 100% des chemins structurels de la fonction sous test.

Nous avons couvert les trois chemins faisables de la fonction sous test en trois cas de test. La soumission du notre graphe mixte au critère TLCM n'a donc pas introduit de cas de test redondant pour la fonction sous test.

13.2.6 Graphe mixte de la fonction sous test et critère TLCS

La soumission du graphe mixte de la fonction sous test au critère TLCS n'améliorera pas les résultats obtenus précédemment car ils sont déjà optimaux pour l'application du critère TLCM. Il y aurait eu redondance des test si plusieurs chemins en sortie de l'appel de la fonction imbriquée étaient faisables pour un même domaine fonctionnel de la fonction `delta_tab` et pour un même contexte d'appel de la fonction imbriquée. Nous n'allons pas comme dans la section précédente détailler les résultats obtenus dans la mesure où ceux-ci seraient similaires aux résultats obtenus par la soumission de notre graphe mixte au critère TLCM.

Les trois soumissions successives de notre graphe mixte à critère TLCS nous amène à exécuter les même 3 cas de test. Les temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux sont : 0.01, 0.01 et 0.01.

	"inlining"	bouchon fonctionnel	TLCM[a]	TLCS[b]
nombre de cas de test	13	1	3	3
% des chemins couverts de `f`	100	33	100	100
temps d'exécution[c]	0.03 0.05 0.05	0.00 0.00 0.01	0.02 0.03 0.02	0.01 0.01 0.01
cas de test redondants pour `f`	10	0	0	0

[a] critère tous-les-chemins-mixtes
[b] critère tous-les-chemins-structurels
[c] temps d'exécution CPU en secondes sur un PC de 2GHZ fonctionnant sous Linux

TAB. 13.2 – Différents résultats obtenus pour le test de la fonction appelant la fonction `delta_tab`

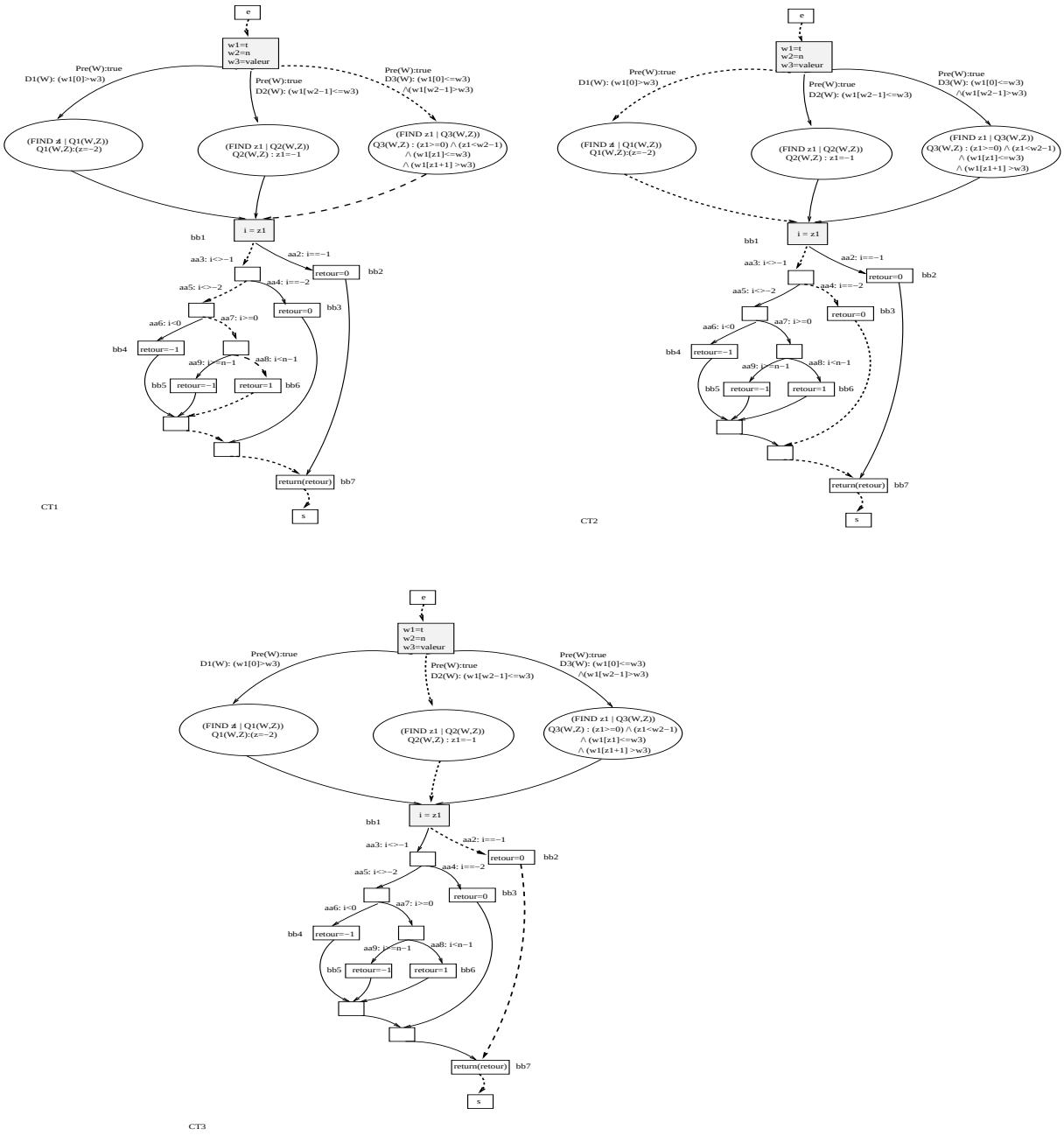

FIG. 13.18 – Chemins couverts lors de la soumission du graphe mixte de la fonction sous test au critère TLCM

13.3 Conclusion

En faisant tourner nos deux exemples, nous avons montré d'une part les inconvénients d'un traitement "inlining" des fonctions imbriquées et d'un traitement par bouchons fonctionnels comme le montre les deux tableaux 13.1 et 13.2 résumant les résultants des différentes méthodes appliquées aux deux fonctions imbriquées utilisées dans nos fonctions sous test. D'un côté, nous avons un traitement "inlining" qui garantit le maintien de la couverture de la fonction appelante mais qui introduit de nombreux cas de test et se heurte au problème de l'explosion combinatoire des chemins et d'un autre côté, l'utilisation de bouchons fonctionnels ne garantit pas le maintien

de la couverture de la fonction appelante.

Notre modélisation des fonctions sous test avec appels sous forme de graphes mixtes a pour objectif de s'inspirer de ces méthodes pour en tirer les avantages respectifs. Sur les précédents exemples, ces graphes mixtes sont manipulés comme un graphe de contrôle et soumis à deux critères distincts à savoir les critères TLCM et TLCS. Nous observons, pour ces deux critères, que l'explosion combinatoire est bien limitée par l'abstraction des chemins internes aux fonctions imbriquées et que la couverture de 100% des chemins faisables de la fonction est maintenue.

L"application du critère TLCM amène à atteindre deux couvertures : une couverture structurelle de la fonction sous test et une couverture fonctionnelle des fonctions imbriquées correspondant à tous les domaines fonctionnels par contexte d'appel. L'avantage de cette stratégie est de pouvoir permettre, parallèlement au test unitaire structurel de la fonction appelante, le test en contexte fonctionnel des fonctions imbriquées. Nous disposons du matériel nécessaire pour mettre ce test en contexte en place : il nous suffit de récupérer les valeurs de sortie de la fonction imbriquée et de les soumettre à la spécification associée pour vérifier leur cohérence.

L'application du critère TLCS au graphe mixte de la fonction sous test permet d'éliminer toute redondance dans la couverture des chemins structurels de la fonction sous test.

Les résultats obtenus sont prometteurs et démontrent la validité de notre approche car nous avons atteint nos objectifs initiaux à savoir un environnement de test réaliste (les fonctions imbriquées sont conservées et exécutées), une limitation de l'explosion combinatoire des chemins par abstraction des chemins internes par les informations issues des spécifications des fonctions imbriquées et le maintien du respect du critère des k-chemins pour la fonction sous test.

Chapitre 14

Conclusion

14.1 Rappels des objectifs

Nos objectifs étaient de tester une fonction sous test contenant des appels dans un environnement le plus proche de la réalité, en limitant l'exploration des fonctions appelées et en garantissant le maintien de la couverture structurelle de la fonction sous test. Notre procédure de test nous permet, en effet, de garantir la couverture de tous les chemins faisables réels de la fonction sous test sans couvrir des chemins qui sont non exécutables en réalité.

Nous sommes partis des deux principales techniques existantes quant à la gestion des appels de fonctions : le traitement "inlining" des appels imbriqués qui permet de conserver un environnement de test réaliste et l'utilisation de bouchons fonctionnels permettant de limiter la combinatoire des chemins par l'abstraction des fonctions imbriquées utilisant leurs spécifications.

14.2 Bilan

Pour notre gestion des appels de fonction, nous disposons du code source de la fonction sous test et des fonctions imbriquées (ou des exécutables des fonctions imbriquées). L'utilisateur nous fournit également la spécification complète et déterministe de chaque fonction appelée.

Comme pour un traitement "inlining", les fonctions appelées sont exécutées dans le corps de la fonction sous test : ce qui nous permet de conserver un comportement à l'exécution très réaliste. Comme l'utilisation de bouchons fonctionnels, nous utilisons les spécifications des fonctions imbriquées pour abstraire les chemins structurels imbriqués, ce qui nous permet de limiter l'explosion combinatoire des chemins.

Les chemins couverts à chaque cas de test contiennent des informations structurelles de la fonction sous test et des informations fonctionnelles des fonctions imbriquées : chaque chemin structurel imbriqué est abstrait par l'expression des contraintes du domaine fonctionnel activé. Nous proposons ainsi une nouvelle modélisation des fonctions sous test contenant des instructions d'appel. Cette modélisation correspond au CFG de la fonction sous test dont les blocs d'appel ont été remplacés par les graphes abstraits des fonctions imbriquées consistant en une représentation fonctionnelle des fonctions imbriquées associée à une mise en correspondance des variables structurelles et fonctionnelles des fonctions. Nous rappelons que les variables fonctionnelles sont les variables abstraites issues de la spécification des fonctions imbriquées et les variables structurelles sont les variables concrètes du code source de la fonction sous test.

Cette modélisation des fonctions sous test est désignée comme le graphe mixte de la fonction sous test pouvant être manipulé comme tout graphe de contrôle. Notre modélisation est générale est peut être reprise par d'autres méthodes de test structurel et soumise aux critères de test associés (tous-les-chemins, toutes-les-branches,...).

Deux nouveaux critères visant à assurer la couverture structurelle des fonctions sous test seront proposés. Le premier critère nommé TLCM pour tous-les-chemins-mixtes correspond à la couverture de tous les chemins de cette nouvelle représentation sous forme de graphe mixte incluant toutes les portions issues de la description fonctionnelle des fonctions appelées. Le second critère TLCS pour tous-les-chemins-structurels vise à la couverture des seules parties structurelles des chemins dans le graphe mixte de la fonction sous test.

L'application du critère TLCM au graphe mixte d'une fonction sous test revient à couvrir tous les k-chemins structurels de la fonction sous test et tous les domaines fonctionnels des fonctions imbriquées pour chaque contexte d'appel. Nous pouvons affirmer que nous limitons l'explosion combinatoire des chemins en se basant sur l'hypothèse raisonnable suivante : pour chaque domaine fonctionnel d'une fonction correspond au moins un chemin structurel. Nous obtenons donc une couverture structurelle de la fonction sous test pour le critère des k-chemins et une couverture fonctionnelle des fonctions imbriquées pour le critère "tous les domaines fonctionnels par contexte d'appel". Un oracle automatique pour les fonctions imbriquées peut être mis en place presque gratuitement : les fonctions imbriquées étant réellement exécutées, il suffit de récupérer les entrées et sorties réelles et de les soumettre à la spécification associée. Il est alors aisé de mettre en place, parallèlement au test structurel de la fonction sous test, une stratégie fonctionnelle de test imbriqué pour les fonctions appelées afin de pouvoir augmenter la confiance associée à ces fonctions. Celles-ci peuvent donc être testées fonctionnellement au sein de leur environnement (évaluées au milieu des autres fonctions du logiciel auxquelles elles appartiennent). Notre objectif étant de limiter l'exploration des fonctions imbriquées au maximum tout en maintenant la couverture structurelle unitaire de la fonction sous test, nous ne nous sommes pas attardés sur la mise en place du test imbriqué des fonctions appelées. Notons cependant qu'il s'agit d'une des perspectives que nous nous sommes fixées à court terme.

L'application du critère TLCS sur le graphe mixte d'une fonction sous test permet de se concentrer sur la couverture de la fonction sous test en limitant au maximum l'exploration des fonctions imbriquées afin d'éliminer les cas de test redondants pour les chemins structurels de la fonction sous test. Le critère TLCS est donc une restriction du critère TLCM à la couverture chemins structurels de la fonction sous test. Le critère TLCS correspond à une modification de la stratégie d'exploration en profondeur d'abord du graphe mixte de façon à ce que les domaines fonctionnels des fonctions imbriquées ne soient couverts que lorsque cela est nécessaire. Pour un contexte d'appel donné d'une fonction imbriquée, différents chemins partiels en sortie dans la fonction sous test peuvent être couverts. Nous forçons la couverture d'un nouveau domaine fonctionnel de la fonction sous test uniquement s'il existe des chemins partiels en sortie n'ayant pu être couverts par le domaine fonctionnel courant de la fonction imbriquée. De plus, pour éviter toutes redondance dans la couverture des chemins structurels de la fonction sous test, nous couvrons un nouveau domaine fonctionnel de la fonction imbriquée en forçant l'exploration des chemins partiels en sortie encore non couvert pour ce contexte d'appel de la fonction imbriquée. Notons que l'application du critère TLCS ne permet pas, contrairement au critère TLCM, la mise en place d'une technique de test imbriqué pour les fonctions imbriquées.

Nous avons validé notre stratégie par la soumission de deux exemples au prototype que nous avons implanté.

Le premier exemple (académique) est une fonction sous test faisant appel à une fonction récursive. Cette fonction récursive possède un très grand nombre de chemins structurels (102 chemins exécutables) mais une spécification simple (deux domaines fonctionnels uniquement). Pour bien illustrer les avantages de notre approche, une fonction sous test contenant exactement 4 chemins structurels exécutables et faisant appel à cette fonction récursive est soumise à un traitement "inlining" (205 cas de test), à un traitement par bouchons fonctionnels (2 cas de test), à une soumission du graphe mixte selon le critère TLCM (6 cas de test) et selon le critère TLCS (4 cas de test). Le gain obtenu par l'abstraction des chemins structurels de la fonction imbriquée via l'expression de ses domaines fonctionnels est immédiat.

Le second exemple est un exemple moins académique : la fonction sous test fait appel à une fonction de linéarisation de fonction. Ce type de fonctions est couramment utilisé en traitement

de signal par exemple. La fonction imbriquée comporte également un nombre élevé de chemins structurels (13 pour notre exemple). La fonction sous test appelant cette fonction imbriquée contient en tout 3 chemins structurels exécutables que nous cherchons à couvrir. Cette fonction sous test est soumise, comme pour l'exemple précédent, à un traitement "inlining" (13 cas de test), à un traitement par bouchon fonctionnel (1 unique cas de test), à une soumission du graphe mixte selon le critère TLCM et selon le critère TLCS (3 cas de test pour les deux derniers cas).

Les expérimentations ont montré des résultats concluants comme le montre les tableaux 13.1 et 13.2 page 201. La soumission du graphe mixte au critère TLCM montre qu'une grande partie des objectifs est atteint : la fonction sous test est évaluée dans un environnement proche de la réalité (sans couvrir des chemins non exécutables en réalité et conservant tous les chemins faisables réels), la couverture de la fonction sous test est maintenue et l'exploration des fonctions imbriquées est bien limitée (à partir du moment où au moins 2 chemins structurels de l'implantation correspondent à un même domaine fonctionnel). De plus, une mise en place d'une technique de test imbriqué pour les fonctions imbriquées est quasi immédiate.

L'application du critère TLCS sur le graphe mixte de la fonction sous test permet d'atteindre tous nos objectifs initiaux : l'exploration des fonctions imbriquées est limitée au maximum c'est-à-dire à l'exploration minimale nécessaire pour maintenir la couverture structurelle des chemins de la fonction sous test. Ainsi, l'application du critère TLCS permet d'éliminer la couverture redondante des chemins structurels de la fonction sous test par rapport à l'application du critère TLCM.

Notre gestion des appels de fonction possède les mêmes avantages que les techniques "inlining" et d'utilisation de bouchons tout en palliant également leurs inconvénients.

14.3 Perspectives

Nous sommes conscients, que pour pouvoir valider définitivement notre approche, nous devons continuer les expérimentations. Nous envisageons de valider la méthode sur d'autres exemples de fonctions plus industrielles (réalistes). Une expérimentation à court terme serait de traiter une fonction imbriquée possédant une spécification disjonctive. Cela nous permettrait également de mettre en place un traitement des contraintes booléennes par l'utilisation de contraintes globales, plus efficaces que des contraintes réifiées.

Notre méthode ne traite pour l'instant que les types entiers, nous envisageons d'intégrer rapidement le traitement de [BGM06] pour la gestion des flottants dans notre méthode de génération automatique de cas de test et dans notre stratégie de gestion des appels. Notons que l'intégration des flottants va augmenter la difficulté d'abstraction et de concrétisation des variables pour notre stratégie de gestion des appels dans la mesure où nous risquons de nous heurter à un problème d'interprétation entre les réels de la spécification et les flottants de l'implantation.

Enfin, nous nous sommes limités à des types C simples dans ce document mais nous désirons passer à une échelle supérieure avec la prise en compte de variables C plus complexes (listes, utilisation des pointeurs dans la spécification ou encore pour la manipulation de listes chaînées, ...) demandant, là encore, un travail supplémentaire en termes d'abstraction et de concrétisation de variables.

Une fois que les précédents objectifs auront été atteints. Notre modélisation des fonctions via notre langage de spécification peut être réutilisée à des fins différentes.

14.3.1 Problème des chemins manquants

En supposant disposer de la spécification de la fonction sous test, nous pouvons mettre en place une couverture structurelle de chaque domaine fonctionnel de la fonction sous test. Le domaine de test est découpé selon ses domaines fonctionnels et la fonction sous test est testée avec le critère des k-chemins successivement sur chacun de ses domaines fonctionnels. Le but est alors de répondre au problème des chemins manquants [GG75], problème adressé uniquement pas des méthodes de test fonctionnel.

14.3.2 Mise en place d'un oracle automatique de la fonction sous test

En demandant à l'utilisateur de fournir également la spécification de la fonction sous test selon les mêmes contraintes que pour les fonctions imbriquées, nous pouvons soumettre les valeurs réelles en entrée et en sortie pour chaque cas de test à la spécification de la fonction sous test. A partir d'une spécification formelle de la fonction sous test, nous pouvons, de façon logique, envisager la mise en place d'un oracle automatique de la fonction sous test en utilisant les postconditions de la spécification comme oracle pour le domaine fonctionnel associé.

14.3.3 Test imbriqué des fonctions imbriquées

Nous en avons déjà parlé précédemment mais la mise en place d'une technique de test imbriqué fonctionnel pour les fonctions imbriquées peut facilement être mise en place dans la mesure où nous disposons du matériel nécessaire. Lors de l'application du critère TLCM sur le graphe mixte d'une fonction sous test, en récupérant les valeurs en entrée et en sortie réelles de la fonctions imbriquées, nous pouvons les soumettre à la spécification fournie par l'utilisateur afin de vérifier automatiquement leur cohérence. Les postconditions de la spécification seront ainsi utilisées comme oracle pour les domaines fonctionnels associés identifiés par les valeurs en entrée.

Toutes ces extensions demandent un effort supplémentaire en termes de concrétisation et abstraction de variables mais cet effort fourni, nous disposons de toutes les briques pour pouvoir mettre en place toutes les orientations que nous venons de citer.

Notre objectif final est un passage à l'échelle au monde industriel. Nous espérons, en effet, que notre travail facilitera la mise en œuvre pratique des techniques d'automatisation de test, tellement prometteuses mais encore trop peu employées à ce jour.

Annexe A

Autres notions utiles du langage C

A.1 Structures conditionnelles Switch et SwitchBreak

Pour une fonction contenant de multiples chemins, l'utilisation imbriquée des structures IfThenElse et IfThen peut compliquer la lecture de la fonction. Une solution souvent usitée est l'emploi de la structure Switch. Celle-ci permet de spécifier un nombre arbitrairement grand de choix entre des exécutions basées sur l'évaluation d'une unique expression. Sa syntaxe est présentée dans la figure A.1.

```
1 switch (exp)
2 {
3  case exp1:b1;
4  case exp2:b2;
5  /*...*/
6  case expn:bn;
7  default:b-default;
8 }
```

FIG. A.1 – Syntaxe C de la structure Switch

Les parenthèses qui suivent le mot réservé `switch` du langage C indiquent un test sur la valeur de l'expression `exp` sur chacun des cas listés ensuite.

Lorsque la valeur de l'expression testée est égale à une des valeurs des expressions de `exp1` à `expn` alors l'ensemble d'instructions associé est exécuté. Par exemple, si une égalité est constatée entre la valeur de `exp` et la valeur de l'expression `exp2` alors les instructions de `b2` sont exécutées. Le mot réservé `default` qui précède l'ensemble d'instructions `b-default` signifie que ce dernier sera exécuté par défaut.

Il faut également préciser que si deux expressions ou plus de `exp1` à `expn` vérifient une égalité avec `exp` alors tous les ensembles d'instructions associés seront exécutés séquentiellement ce qui implique alors une exécution systématique des instructions de `b-default`. Pour éviter cela, nous pouvons utiliser une structure dérivée, la structure SwitchBreak utilisant l'instruction `break` qui provoque la sortie de la structure SwitchBreak après l'exécution d'un bloc d'instructions associé à un `Case`. Cette instruction ajoutée à la fin des ensembles d'instructions de chaque `Case` (cf. figure A.2) implique que seul un ensemble d'instructions sera exécuté y compris l'ensemble d'instructions par défaut.

Ainsi, pour une structure SwitchBreak possédant n `Case` différents alors $n+1$ flots de contrôle sont créés.

```
switch (exp)
{
 case exp1:b1;break;
 case exp2:b2;break;
 /*...*/
 case expn:bn;break;
 default:b-default;
}
```

FIG. A.2 – Syntaxe C de la structure Switch avec l'utilisation de l'instruction `break`

A.2 Structures répétitives ForDo et DoWhile

Structure DoWhile

La seconde structure répétitive, DoWhile, se rapproche beaucoup de la précédente. Sa syntaxe est en effet assez proche (cf. figure A.3).

```
do
 b1;
while  (exp)
b;
```

FIG. A.3 – Syntaxe C de la structure DoWhile

La sémantique de la structure DoWhile consiste en une première exécution systématique de l'ensemble d'instructions `b1` qui sera de nouveau exécuté tant que l'expression `exp` est vérifiée.

La différence notable entre ces deux structures répétitives est donc que la première, While, entraîne un nombre d'exécutions de l'ensemble des instructions de la boucle allant de 0 à l'infini alors que la seconde, DoWhile, implique dans tous les cas au minimum une exécution de l'ensemble d'instructions de la boucle.

Structure ForDo

La structure répétitive suivante, ForDo, est une boucle à nombre d'itérations fixé. La figure A.4 est une illustration de sa syntaxe.

```
for (var=exp;cond;delta-var)
 b1;
b;
```

FIG. A.4 – Syntaxe C de la structure ForDo

Son nombre d'itérations est calculé à partir d'une borne initiale (en général la valeur de départ d'une variable) `"var=exp ;"`, d'une borne finale caractérisée par une condition d'arrêt de cette boucle (en général une condition `cond` portant sur la valeur de la variable `var`) et d'une variation de cette variable à chaque itération de la structure `delta-var` qui consiste le plus souvent à incrémenter ou décrémenter la valeur de la variable `var` à chaque passage dans la boucle. La sémantique de l'instruction est la suivante :

- la variable `var` est d'abord initialisée avec la valeur de l'expression `exp` puis
- tant que l'expression `cond` est vérifiée alors :

1. l'ensemble d'instructions `b1` est exécuté puis
2. l'instruction de variation `delta-var` est exécutée.

Ainsi, si l'instruction de départ de la boucle `"var=exp ;"` implique que la valeur de la variable `var` ne vérifie pas l'expression `cond` alors les instructions de `b1` ne seront jamais exécutées.

Illustration 80
Prenons le cas suivant :

- *`for(i=0 ;i<10 ;i++)` exécute 10 fois le bloc d'instructions interne à la boucle (`i` varie de 0 à 9)*

Annexe B

Rappels sur la théorie des graphes

Les méthodes de test structurel induisent une représentation graphique de la structure interne (l'implantation) du programme sous test. Cette représentation appelée graphe de flot de contrôle (cf. chapitre 4) requiert donc, avant toute description détaillée, de donner quelques notions nécessaires de la théorie des graphes (cf. [Ber58]).

B.1 Classe de graphes considérée de la théorie des graphes

Nous considérons ici une classe particulière de graphes connexes orientés étiquetés ayant un unique nœud d'entrée e et un unique nœud de sortie s que nous noterons $G =< N, E, e, s, \delta >$.

Nous allons expliciter les éléments de ce quintuplet.

Définition – B.1.1

*Un **graphe orienté** $G =< N, E >$ où N est un ensemble fini et E une relation binaire sur N. L'ensemble N est l'ensemble des nœuds de G et E l'ensemble des arcs de G.*

Les graphes ont une représentation graphique communément admise où les nœuds seront ici représentés par des rectangles et les arcs par des flèches, tel que si (u, v) est un arc d'un graphe orienté on dit que (u, v) a pour origine le nœud u et pour cible le nœud v.

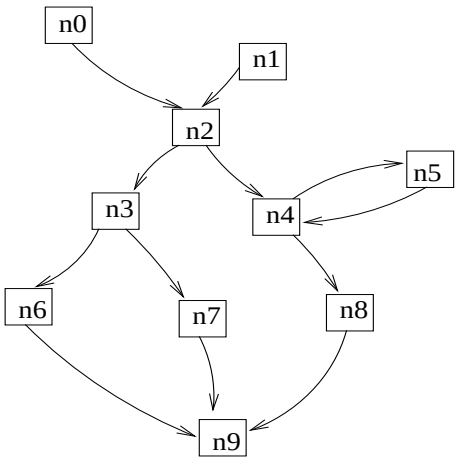

Fig. B.1 – Illustration d'un graphe orienté $G =< N, E >$

Illustration 81
Prenons la figure B.1. Le graphe $G =< N, E >$ *est un graphe orienté avec :* $N = \{n0, n1, n2, n3, n4, n5, n6, n7, n8, n9\}$
$E = \{(n0, n1), (n1, n2), (n2, n3), (n2, n4), (n3, n6), (n3, n7), (n4, n5), (n4, n8),$
$(n5, n4), (n6, n9), (n7, n9), (n8, n9)\}$
Si on prend au hasard l'arc $(n2, n3)$, $n2$ *est l'origine et* $n3$ *la cible de cet arc.*

L'étiquetage des graphes permet d'associer des valeurs (noms, condition, calculs ...) aux différents arcs et nœuds du graphe.

Définition – B.1.2

Un ***graphe étiqueté*** *est un graphe G=<N,E> dont les arcs et les nœuds sont affectés d'étiquettes définies par la fonction d'étiquetage* δ *suivante :*

$$\delta : (\delta_N, \delta_E)$$

$$\delta_N : N \rightarrow L_N$$

$$\delta_E : E \rightarrow L_E$$

avec L_E *(resp.* L_N*) l'ensemble des étiquettes associées à l'ensemble* E *(resp.* N*).*

Notation 22
Nous noterons l'ensemble des graphes étiquetés $G =< N, E, \delta >$ *avec* δ *la fonction d'étiquetage associée.*

Nous utiliserons également par la suite la notion de successeurs et prédécesseurs de nœud d'un graphe.

Définition – B.1.3

Les ***prédécesseurs*** *d'un nœud n, notés pred(n), sont l'ensemble des nœuds liés à n par un arc dont n est la cible. Ainsi on a pred(n)=*$\{m \in N | (m, n) \in E\}$*. Réciproquement les* ***successeurs*** *d'un nœud n, notés succ(n), sont l'ensemble des nœuds liés à n par un arc dont n est l'origine. Ainsi on a succ(n)=*$\{o \in N | (n, o) \in E\}$*.*

Illustration 82
Toujours avec la figure B.1. Prenons les deux nœuds du graphe $n2$ *et* $n3$ *:*

$$pred(n2) = \{n0, n1\}$$

$$pred(n3) = \{n2\}$$

$$succ(n2) = \{n3, n4\}$$

$$succ(n3) = \{n6, n7\}$$

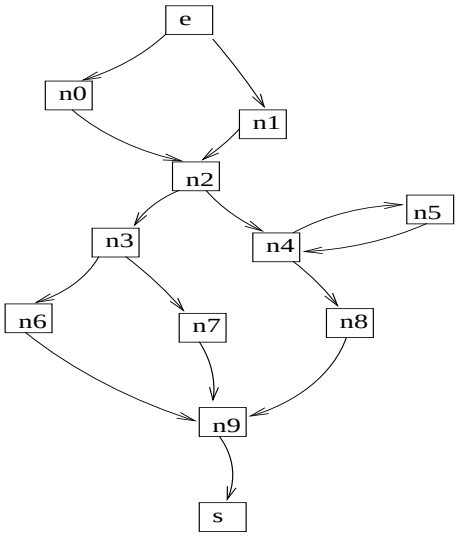

FIG. B.2 – $G =< N, E, e, s >$: graphe orienté à unique nœud d'entrée et unique nœud de sortie

DÉFINITION – B.1.4

*Un **graphe avec un unique nœud d'entrée** e **et un unique nœud de sortie** s $G =< N, E, e, s >$ est un graphe construit à partir d'un graphe $G =< N', E' >$ tel que :*

$$\forall n \in N, pred(n) = \emptyset, N = N' \cup \{(e, n)\}$$

$$\forall n \in N, succ(n) = \emptyset, N = N' \cup \{(n, s)\}$$

$$E = E' \cup \{e, s\}$$

Illustration 83
Le résultat de la construction du graphe $G =< N, E, e, s >$ à partir du graphe $G =< N, E >$ de la figure B.1 se trouve dans la figure B.2.

B.2 Les notions de chemins pour $G =< N, E, e, s, \delta >$

DÉFINITION – B.2.1

*Dans un graphe orienté $G =< N, E, e, s, \delta >$ un **chemin total** noté $Ch(e, s)$ est une séquence $(e, (e, n_1), n_1, \ldots, (n_{k-1}, n_k), n_k, (n_k, s), s)$ de nœuds et d'arcs telle que $(n_{i-1}, n_i) \in E$ pour tout $i \in \{1..k\}$ et $n_j \in N$ pour tout $j \in, \{1..k\}$ avec $e = n_0$ et $s = n_{k+1}$.*

Illustration 84
Le chemin $Ch(e, s)$ tel que :

$$Ch(e, s) = (e, (e, n1), n1, (n1, n2), n2, (n2, n3), n3, (n3, n7), n7, (n7, n9), n9, (n9, s), s)$$

est un chemin total du graphe de la figure B.2.

Définition – B.2.2

*Dans un graphe orienté $G =< N, E, e, s, \delta >$, un **chemin partiel** noté $Ch_p(n_i, n_j)$, est une séquence $(n_i, (n_i, n_{i+1}), n_{i+1}, \ldots (n_{j-1}, n_j), n_j)$ de nœuds et d'arcs telle que $(n_k, n_{k+1}) \in E$ pour tout $k \in \{i..j-1\}$ et $n_k \in N$ pour tout $k \in \{i..j\}$ avec $n_i \neq e \vee n_j \neq s$.*

Un chemin partiel est également un sous-chemin d'un chemin total.

Définition – B.2.3

*Un chemin $Ch(n_i, n_j)$ est un **sous-chemin** du chemin total $Ch(e, s) = (e, (e, n_1), n_1, \ldots, (n_k, s), s)$ avec $e = n_0$ et $s = n_{k+1}$ si la séquence de $Ch(n_i, n_j)$ est une sous-séquence contiguë des nœuds et des arcs de $Ch(e, s)$ vérifiant : $Ch(n_i, n_j) \subset Ch(e, s)$.*

Illustration 85

Le chemin $Ch(n1, n3)$ tel que :

$$Ch(n1, n3) = (n1, (n1, n2), n2, (n2, n3), n3)$$

est un chemin partiel de $G =< N, E, e, s, \delta >$ et un sous-chemin du chemin total $Ch(e, s)$ donné dans l'illustration 84.

B.3 Connexité et sous-graphe

La catégorie des graphes $G =< N, E, e, s, \delta >$ étudiée correspond à des graphes connexes. Nous pouvons définit le propriété de la connexité.

Définition – B.3.1

*Un graphe $G =< N, E, e, s, \delta >$ est dit **connexe** s'il vérifie la relation suivante :*

$$\forall n \in N, \exists Ch(e, n) \wedge \exists Ch(n, s)$$

avec $Ch(e, n)$ et $Ch(n, s)$ deux chemins partiels de $G =< N, E, e, s, \delta >$.

Illustration 86

Le graphe $G =< N, E, e, s, \delta >$ de la figure B.2 vérifie la propriété de connexité.

La dernière définition sur laquelle nous allons nous attarder est celle d'un sous-graphe.

Définition – B.3.2

*On dit qu'un graphe $G' =< N', E', e', s', \delta' >$ est un **sous-graphe** de $G =< N, E, e, s, \delta >$ si $N' \subseteq N \wedge E' \subseteq E'$.*

Illustration 87

Le graphe $G' =< N', E', e', s', \delta' >$ de la figure B.3 est un sous-graphe de $G =< N, E, e, s, \delta >$ de la figure B.2. Notons que ce sous-graphe G' n'est en revanche pas un graphe connexe : il n'existe en effet aucun chemin partiel allant du nœud $n5$ au nœud de sortie s.

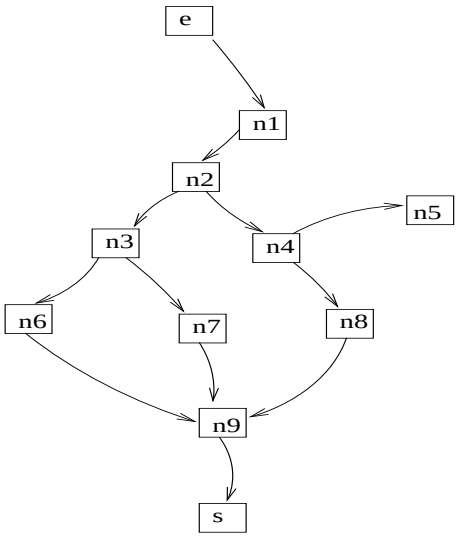

FIG. B.3 – $G' =< N', E', e', s', \delta' >$: sous-graphe de $G =< N, E, e, s, \delta >$ de la figure B.2

Annexe C

Notions de la programmation logique avec contraintes

Cet annexe a pour objectif de présenter la programmation logique avec contraintes appelée aussi PLC ou CLP. Cet outil est souvent utilisé pour des méthodes de test (cf. chapitre 5) pour la détermination de données de test ou aussi, comme dans notre méthode de génération de cas de test unitaire, pour la stratégie de sélection des cas de test (cf. chapitre 7).

C.1 Motivations de la programmation logique avec contraintes

L'objectif de la programmation logique avec contraintes est la résolution d'un ensemble de contraintes. Les contraintes sont spécifiques au problème posé, elles doivent le décrire pour qu'on puisse appliquer le principe de la PLC afin de le résoudre.

La PLC représente une manière puissante et efficace pour résoudre plusieurs problèmes comme par exemple l'optimisation ou pour le cas qui nous intéresse la génération de cas de test logiciel comme dans [GBR00], [MA00],[JM94],[SD01] ou encore [WMM04a].

C.2 Quelques définitions préliminaires

Une **contrainte** est une propriété exprimant une relation (égalités, inégalités,...) entre différentes variables à vérifier.

Chaque variable prend ses valeurs dans un ensemble donné que l'on appelle **domaine**. Selon le domaine considéré, il peut être décrit soit sous la forme d'un ensemble de valeurs soit par un intervalle ou une union d'intervalles. Dans ce mémoire, nous ne considérons que les domaines finis ("Finite Domain"). Ainsi, une contrainte peut être vue comme une restriction des valeurs que peuvent prendre simultanément les variables.

La résolution de contraintes non linéaires sur des réels est un problème indécidable mais, dans ce mémoire, nous ne manipulons que des contraintes sur des domaines finis dont la résolution est un problème NP-complet au pire (contraintes solubles en temps polynomial par rapport au nombre de variables du CSP et à la taille des domaines associés).

Un problème de satisfaction de contraintes CSP ("Constraint Solving Problem") est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables.

Définition – C.2.1

*Un **CSP** est un triplet (V, D, C) tel que :*
- *$V = \{v_1, \ldots, v_n\}$ est un ensemble fini de variables,*
- *$D = \{D_{v_1}, \ldots, D_{v_n}\}$ est un ensemble fini de domaines tel que D_{v_i} est le domaine contenant toutes les valeurs possibles de la variable v_i,*
- *$C = \{c_1, \ldots, c_m\}$ est un ensemble fini de contraintes avec chaque contrainte c_i portant sur un sous-ensemble de V.*

Illustration 88

Le problème des N-Dames consiste à placer N dames sur un échiquier $N * N$ sans qu'aucune des lignes, colonnes et diagonales de l'échiquier ne possède plus d'une dame.

Le CSP associé à ce problème peut être représenté avec N variables $v_{i_{i=1..N}} \in [1..N]$ représentant la position en colonne de la dame de la ligne i. En effet, comme deux dames ne peuvent pas être sur la même ligne, et qu'il y a autant de dames que de lignes, il y a exactement une et une seule dame par ligne. L'ensemble des contraintes C associées aux variables v_i sont, pour tout $i \neq j$ où $j \in [1..N]$:
- *$v_i \neq v_j$ (ne pas mettre deux dames sur la même colonne),*
- *$v_i - i \neq v_j - j$ (ne pas mettre deux dames sur la même diagonale NO-SE),*
- *$v_i + i \neq v_j + j$ (ne pas mettre deux dame sur la même diagonale NE-SO).*

Définition – C.2.2

*Une **affectation** consiste à instancier certaines variables par des valeurs de leur domaine.*

*Une **affectation est dite totale** si elle instancie toutes les variables du CSP et **partielle** si elle instancie qu'une partie des variables du CSP.*

Définition – C.2.3

*Une affectation **satisfait** (resp. **viole**) une contrainte c_i si la contrainte c_i est vérifiée (resp. n'est pas vérifiée) sur l'ensemble des valeurs des variables de c_i.*

Définition – C.2.4

*Une affectation est dite **consistante** si elle ne viole aucune contrainte et est dite **inconsistante** si elle viole au moins une des contraintes du CSP.*

Définition – C.2.5

*Une **solution d'un CSP** correspond à une affectation totale consistante c'est-à-dire quand toutes les valeurs des domaines des variables du CSP vérifient l'ensemble des contraintes du CSP.*

Définition – C.2.6

*La **consistance d'arc** est une méthode très employée qui s'applique dans le cas de contraintes binaires (i.e. impliquant deux variables). Une contrainte binaire peut être représentée par un arc reliant les deux variables impliquées ce qui justifie la terminologie utilisée. Une contrainte satisfait la consistance d'arc si pour chaque valeur d'une des variables, il existe au moins une valeur de l'autre variable appartenant à une solution de la contrainte. On établit la consistance d'arc en supprimant les valeurs qui ne satisfont pas cette propriété.*

Illustration 89

Soient les variables $v_1 \in [1, 2, 3]$ et $v_2 \in [1, 3]$ et la contrainte $v_1 = v_2$. La valeur 2 de v_1 n'appartenant pas à une solution, on la supprime donc du domaine : $v_1 \in [1, 3]$ et $v_2 \in [1, 3]$.

La consistance d'arc est une propriété très forte. Lorsque les domaines des variables sont trop grands, l'énumération de toutes les possibilités devient alors trop lourd. On ne travaille alors plus que sur les bornes inférieures et supérieures du domaine.

Définition – C.2.7

*Une contrainte satisfait la **consistance de bornes** si la valeur minimale et la valeur maximale du domaine des variables de la contraintes vérifient cette contrainte. Un CSP vérifie la consistance de bornes si toutes ces contraintes sont consistantes de bornes.*

Illustration 90

Soient les variables $v_1 \in [1, 2, 3]$ et $v_2 \in [5, 6, 7, 8]$ et la contrainte $v_1 < v_2$. Les valeurs 1 et 3 de v_1 et 5 et 8 de v_2 vérifient la contrainte, la contrainte est donc consistante de bornes.

C.3 Résolution des CSP

Deux des principales méthodes de résolution sont :

- la **simplification des problèmes** qui consiste à transformer le CSP en un problème plus simple à résoudre ou dont la satisfiabilité est connue en utilisant des techniques de consistance c'est-à-dire sur une propagation des valeurs avec réduction des domaines associés aux variables,
- la **recherche de solutions** qui consiste à choisir pour chaque variable du CSP une valeur : on parle de phase de **"labelling"** de façon à obtenir une affectation totale consistante. En cas de violation d'une des contraintes du système, les affectations de variables peuvent être remises en cause en commençant par la dernière variable instanciée : on parle de phase de **"backtracking"** (cf. section C.3.2). Ainsi, la recherche s'arrête lorsqu'une solution est trouvée au CSP (affectation totale consistante), lorsque la recherche de solution échoue (toutes les combinaisons d'affectations de valeurs aux variables ont été tentées et étaient inconsistantes) ou encore lorsque la limitation d'une ressource allouée au temps de recherche est atteinte (par exemple, un TimeOut, un nombre donné d'échecs, ...).

Il existe de nombreuses manières d'optimiser la résolution d'un problème CSP. Différents algorithmes de résolution de CSP existent mais adressent un type de CSP donné permettant d'obtenir la ou les solutions de ce CSP. Ainsi, pour chaque type de CSP, il existe des heuristiques (de labelling par exemple) et également des solveurs de contraintes adaptés.

En ce qui nous concerne, nous avons désiré être plus génériques et ne pas traiter uniquement un type de CSP donné. La recherche de solutions étant un problème NP-complet, nous utilisons différentes heuristiques alternativement avec la mise en place d'un TimeOut.

Nous présentons brièvement dans cette annexe les deux premières méthodes de résolution citées précédemment.

C.3.1 Simplification du problème

Cette technique seule permet rarement de résoudre le problème mais est utilisée en alternance avec la phase de "labelling".

Définition – C.3.1

*Deux **CSP équivalents** possèdent le même ensemble V de variables et le même ensemble S de solutions.*

Définition – C.3.2

*Un CSP (V, D, C) est **réduit** en un CSP (V_0, D_0, C_0) si :*
- *les CSP (V, D, C) et $(V_0, D_0, C0)$ sont équivalents,*
- *pour chaque variable v_i de V (et par conséquent également de V_0) le domaine associé dans $(V_0, D_0, C0)$ est un sous-domaine du domaine associé dans (V, D, C),*
- *toute instanciation partielle satisfaisant C_0 satisfait aussi C.*

Un moyen simple de réduire un CSP est d'identifier dans les domaines des variables du système toutes les valeurs dites inutiles (qui ne peuvent pas appartenir à une solution du CSP) en utilisant la méthode de consistance d'arc sur les contraintes binaires du système.

L'efficacité de la recherche de solution avec "backtracking" peut être améliorée si l'on peut couper les branches de l'arbre de recherche sans solution. C'est précisément sur ce point que la simplification de problème peut aider : la réduction de la taille du domaine d'une variable revient à couper certaines branches. La simplification du problème peut être utilisée à n'importe quel moment lors de la recherche de solution.

Remarque(s) 38

Plus un CSP peut être simplifié et plus le coût de calcul est important lors de la phase de simplification. Parallèlement, plus un CSP à été simplifié et moins la phase de labelling sera coûteuse lors de la recherche de solutions.

Il faut donc trouver un bon compromis entre les efforts à faire et le gain découlant de la phase de simplification du CSP.

Ainsi, les techniques de simplification transforme un CSP en un CSP équivalent simplifié c'est-à-dire avec des domaines de taille réduite.

C.3.2 Recherche de solutions

Backtracking

L'algorithme de base pour la recherche de solutions est le "backtracking" appelé aussi algorithme de simple retour-arriere. Il s'agit d'une stratégie générale de recherche largement utilisée pour la résolution de problèmes.

Lors du "labelling", si la valeur choisie pour une variable viole au moins une contrainte alors au moins cette contrainte ne peut être satisfaite. Dans un tel cas, l'algorithme du "backtracking" consiste à choisir une nouvelle valeur pour la dernière variable instanciée

Remarque(s) 39

Notons que le "backtracking" est un mécanisme de la programmation logique (avec ou sans contraintes) qui peut être utilisé par l'algorithme de "labelling". L'algorithme de "backtracking" peut amener à changer la valeur de toutes les variables assignées du système par des retour-arrières successifs.

Les deux étapes dans l'algorithme de "backtracking" sont le choix de la prochaine variable à instancier et de la valeur à lui affecter.

Choix d'une variable à instancier

Il existe quatre principales heuristiques de choix de variables à instancier :
- la stratégie "fail first principe" (principe de l'échec d'abord) qui consiste à instancier les variables par ordre croissant de domaine,
- la stratégie "minimal width ordering" (ordonnancement selon la largeur minimale) qui consiste à instancier d'abord les variables selon le nombre de variables dont elles dépendent,
- la stratégie "minimal bandwidth ordering" (ordonnancement selon la cardinalité minimale) qui consiste à instancier d'abord une variable au hasard puis la variable qui partage le plus de contraintes avec la variable précédemment instanciée,

– la stratégie "maximum cardinality ordering" (ordonnancement selon le degré maximal) qui consiste à instancier d'abord une variable de CSP puis de choisir une variable apparaissant dans le plus grand nombre de contraintes portant sur les variables déjà instanciées.

Les trois dernières heuristiques sont des ordonnancements statiques car ils sont effectués en amont de la procédure de recherche contrairement à la première méthode effectuée dynamiquement et donc profitant de la simplification du CSP (par propagation de contraintes) faite lors de chaque instanciation de variables.

Choix d'une valeur

Une fois la variable à instancier choisie, il faut lui affecter une valeur de son domaine.

Les heuristiques de choix de valeur sont nombreuses. Une heuristique que nous pouvons citer est celle des valeurs les plus contraignantes ("most-constraining-value"). Cela consiste à choisir la valeur du domaine d'une variable qui va avoir le plus de conséquences pour le choix des valeurs des autres variables c'est-à-dire la valeur qui va provoquer le plus de valeurs redondantes par propagation de contraintes. Il s'agit pour cela de faire une analyse assez détaillée des contraintes et des domaines des variables associées afin de déterminer quelles valeurs d'une variable donnée vont entraîner la plus grande diminution des domaines des autres variables du système lors de la propagation de contraintes. L'idée est de pouvoir détecter une insatisfiabilité au plus tôt. Cela correspond donc à simuler les phases de "labelling" et de simplification pour identifier les valeurs les plus contraignantes du domaine de la variable que l'on cherche à instancier.

D'autres stratégies de choix de valeurs s'orientent, par exemple, vers le choix d'une valeur minimale (ou maximale) du domaine de la variable à instancier. Nous pouvons aussi parler de du choix de la valeur utilisant la bissectrice du domaine qui consiste à choisir une valeur à droite ou à gauche de cette bissectrice et en cas d'insatisfiabilité, de supprimer une moitié du domaine de la variable à instancier.

Les heuristiques de choix de valeurs efficaces (en termes de limitation de "backtrack") nécessite un travail coûteux d'analyse des contraintes en amont du "labelling".

C.4 Les solveurs de contraintes

Remarque(s) 40
Nous ne parlerons dans cette section uniquement de solveurs de contraintes pour la programmation logique.

Les solveurs de contraintes sont des logiciels complexes pouvant être représentés par un algorithme de recherche combiné avec un algorithme d'itération d'opérateurs de réduction de domaines. Les solveurs de contraintes sont généralement construits à partir de librairies (comme par exemple celle des domaines finis ou des contraintes linéaires sur les réels) et de mécanismes provenant soit d'environnement de programmation par contraintes comme ECLiPSe [WNJ97] soit de mécanismes supplémentaires implantés pour augmenter l'efficacité de la résolution.

Un solveur de contraintes est un logiciel ayant pour objectif de résoudre un système composé de contraintes arithmétiques (y compris min, max,...) et/ou de contraintes globales ($alldifferent$,...) par l'utilisation de différents mécanismes comme les prédicats de choix de variables et de valeurs de variables, la propagation de contraintes et l'unification entre autres.

L'unification de deux termes consiste à rechercher la substitution minimale pour les variables des termes rendant ceux-ci égaux.

Les contraintes booléennes se traduisent en opérations booléennes qui seront traitées par unifications et simplifications symboliques.

Les contraintes numériques portent sur des variables à valeurs numériques. Une contrainte numérique est une différence, une égalité ou une inégalité entre 2 expressions arithmétiques. Ces

contraintes peuvent être définies sur les entiers, les variables dans ce cas peuvent prendre des valeurs entières mais aussi sur les réels quand les variables de la contrainte peuvent prendre des valeurs réelles. Les contraintes numériques sont linéaires quand les expressions arithmétiques sont linéaires (par exemple, $2x + y = 0$) ou non linéaires quand les expressions arithmétiques sont de degré supérieur à 1 (par exemple pour un produit de variables) ou qu'elles contiennent des fonction exponentielles, logarithmiques, ...

Le principe d'un solveur de contraintes peut être décrit en quatre étapes principales :

- simplifier le problème donné, puis stocker les contraintes pas encore satisfaites dans un magasin de contraintes jusqu'à atteindre une affectation permettant de faire évoluer la recherche de solutions du CSP,
- choisir puis instancier une variable selon la (les) heuristique(s) programmée(s),
- l'affectation d'une variable entraîne la propagation des contraintes associées qui peuvent :
  - disparaître si elles sont résolues ou
  - activer d'autres contraintes ou enfin
  - amener à un échec de la résolution du CSP.
- quand toutes les contraintes ont été traitées alors :
  - pour une affectation totale consistante : succès de la recherche de solutions et arrêt,
  - pour une affectation inconsistante : "backtracking" et recherche d'une nouvelle affectation totale du CSP,
  - pour une affectation totale inconsistante et "backtracking" impossible alors arrêt sur un échec de résolution du CSP.

Bibliographie

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT : A tool-set for test generation from Z and B using constraint logic programming. In *Proc. of Formal Approaches to Testing of Software, FATES 2002 (workshop of CONCUR'02)*, pages 105–120, Brnö, République Tchèque, August 2002. INRIA report.

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Breckner, P. D. Mosses, D. Sannella, and A. Tarlecki. Casl : the common algebraic specification language. *Theor. Comput. Sci.*, 286(2) :153–196, 2002.

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments ? In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[ABM97] L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING : A formally based software test generation method. In *ICFEM'97, First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japon, November 1997.

[Abr96] J.-R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.

[Ame86] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Draft Proposed American National Standard Programming Language C*, October 1 1986.

[BCH+04] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples, 2004.

[BDL06] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006 : Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443. Springer, 2006.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New-York, 1990.

[Ber58] C. Berge. *Théorie des graphes et ses applications.* Collection Universitaire des Mathématiques, Dunod, Paris, 1958.

[BGM06] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations : Research articles. *Softw. Test. Verif. Reliab.*, 16(2) :97–121, 2006.

[BHJT00] B. Baudry, V. Hanh, J. Jezequel, and Y. Traon. Trustable components : Yet another mutation-based approach, 2000.

[BORZ99] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess : a specification-driven testing environment for synchronous software. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 267–276, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system : An approach to testing based on combinatiorial design. *Software Engineering*, 23(7) :437–444, 1997.

[CE05] C. Cadar and D. R. Engler. Execution generated test cases : How to make systems code crash itself. In *SPIN*, pages 2–23, 2005.

[CFK+91] R. Cytron, J. Ferrante, B. K.Roseb, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependance graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :pp. 451–490, October 1991.

[CL93] R. Cori and D. Lascar. *Logique mathématique : calcul propositionnel, algèbres de Boole, calcul des prédicats*. Collection de loqique mathématique AXIOMES, Masson, Paris, 1993.

[CL01] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing : The JML and JUnit way. Technical Report 01–12, 2001.

[CLRZ99] A. R. Cavalli, D. Lee, C. Rinderknecht, and F. Zaidi. Hit-or-jump : An algorithm for embedded testing with applications to IN services. In *FORTE*, pages 41–56, 1999.

[Cow91] P. D. Coward. Symbolic execution and testing. *Information and Software Technology*, 33(1) :53–64, 1991.

[CR85] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *J. Syst. Softw.*, 5(1) :15–35, 1985.

[CS94] J. Chilenski and S.Miller. Applicability of modified condition /decision coverage to software testing. *Software Engineering Journal*, pages 193–200, 1994.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Springer Verlag, editor, *FME93 Industrial-Strength Formal Methods*, volume LNCS 670, pages pp. 268–284, FME Europe, April 1993. J.C.P. Woodcock and P.G. Larsen.

[DGM93] P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing : A case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3) :229–244, 1993.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11(4) :34–41, 1978.

[drV06] Projet de recherche V3F. *Validation et Vérification en présence de calculs à Virgule Flottante*. Sécurité informatique, 2005/2006. http ://lifc.univ-fcomte.fr/~v3f/.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive inter-procedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages pp. 242–256, june 1994.

[Fag] M. E. Fagan. Design and code inspection and process control in the development of programs.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV*, pages 348–359, 1996.

[FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1) :63–86, 1996.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10) :1483–1498, 1988.

[GA95] P. Le Gall and A. Arnould. Formal specifications and test : Correctness and oracle. In *COMPASS/ADT*, pages 342–358, 1995.

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. *Lecture Notes in Computer Science*, 1861 :399–413, July 2000.

[GBW06] A. Gotlieb, B. Botella, and M. Watel. Inka : Ten years after the first ideas. In *Proc. ICSSEA 2006*, Paris, December 2006.

[GDGM01] S. Gouraud, A. Denise, M. Gaudel, and B. Marre. A new way of automating statistical testing methods. 2001.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2) :156–173, 1975.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, 2005.

[GMS98] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation. In *Foundations on software engineering*, pages 231–244, 1998.

[GMSB96] M.-C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. *Précis de génie logiciel.* Masson, Paris, 1996.

[GN97] M. J. Gallagher and V. Lakshmi Narasimhan. Adtest : A test data generation suite for ada software systems. *IEEE Trans. Softw. Eng.*, 23(8) :473–484, 1997.

[God07] P. Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1) :47–54, 2007.

[Got00] A. Gotlieb. *Génération automatique de cas de test structurel avec la programmation logique avec contraintes.* PhD thesis, Université de Nice-Sophia Antipolis, janvier 2000.

[Gou04] S.-D. Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique.* PhD thesis, Université Paris XI, Orsay, 2004.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94 : Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :pp. 567–580, 1969.

[JE94] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9) :30–38, 1994.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–581, 1994.

[JO95] Z. Jin and A. J. Offutt. Integration testing based on software couplings. In *Compass '95 : 10th Annual Conference on Computer Assurance*, pages 13–24, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[Jor85] P. C. Jorgensen. *The use of MM-paths in constructive software development.* PhD thesis, Arizona State University, 1985.

[Kin76] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7) :385–394, 1976.

[KL85] B. Korel and J. Laski. A tool for data flow oriented program testing. In *Proceedings of the second conference on Software development tools, techniques, and alternatives*, pages 34–37, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8) :870–879, August 1990.

[KR04] B. W. Kernighan and D. M. Ritchie. *Le langage C Norme ANSI, 2ème édition.* Editions Dunod, 2004. http ://manju.cs.berkeley.edu/cil/.

[KWF92] B. Korel, H. Wedde, and R. Ferguson. Dynamic method of test data generation for distributed software. *Inf. Softw. Technol.*, 34(8) :523–531, 1992.

[LAB+95] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, and P. Thévenod. *Guide de la sûreté de fonctionnement.* Cépaduès-Editions, 1995.

[Lan06] CIL : C Intermediate Language. *CIL - Infrastructure for C Program Analysis and Transformation.* 2005/2006. http ://manju.cs.berkeley.edu/cil/.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In *FME 2002*, volume 2391, pages pp 21–40, In L.-H. Eriksson and P. Lindsay, July 2002. Formal Methods Europe.

[LSKP96] D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach. *IEEE Trans. on Communications*, 44(5) :631–640, May 1996. An early version with a title *Conformance Testing of Protocols Specified as Communicating FSMs,* appeared in *IEEE INFOCOM'93*, March 30 - April 1, pp. 115-127, 1993.

[LW90] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Conference on Software Maintenance-1990*, pages 290–301, San Diego, Nov 1990. CA.

[LY00] J.-C. Lin and P.-L. Yeh. Using genetic algorithms for test case generation in path testing. In *ATS '00 : Proceedings of the 9th Asian Test Symposium*, pages 241–256, Washington, DC, USA, 2000. IEEE Computer Society.

[MA00] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *Proc. ASE 2000*, pages pp 229–237, Grenoble, September 2000. IEEE Computer Society Press.

[MC94] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 132–141, Standford, California, USA, 1994. Springer-Verlag.

[McC96] A. Watson. T. McCabe. Structured testing : A testing methodology using the cyclomatic complexity metric, Aug. 1996.

[Meu01] C. Meudec. ATGen : automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2) :pp 81–96, June 2001.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer*, 40-51(10), October 1992.

[MM98] C. C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Automated Software Engineering*, pages 136–146, 1998.

[Mou03] P. Mouy. Génération automatique de cas de test avec critère structurel de couverture. Rapport dea, Université de Technologie de Compiègne, 2003.

[Mou04] P. Mouy. Vers une méthode de génération de test boîte grise "à la volée". In *Proc. AFADL 2004*, pages 169–183, Besançon, France, juin 2004.

[Mou07] P. Mouy. Application de critères structurels en présence d'appels de fonctions pour la sélection et génération de tests. In *Proc. AFADL 2007*, Namur, Belgique, juin 2007.

[MS04] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Control*, 12(2) :121–136, 2004.

[Mye78] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Commun. ACM*, 21(9) :760–768, 1978.

[Mye79] G. J. Myers. *The Art of Software Testing.* Business Data Processing, Editors : Richard G. Canning and J. Daniel Cougar. WILEY, 1979.

[NFTJ06] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. Automatic test generation : A use case driven approach. *IEEE Trans. Software Eng*, 32(3) :140–155, 2006.

[OAL06] J. Offutt, P. Ammann, and L. (Ling) Liu. Mutation testing implements grammar-based testing. *mutation*, 0 :12, 2006.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6) :676–686, 1988.

[OHK93] A. Jefferson Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *J. Syst. Softw.*, 20(3) :295–308, 1993.

[OVP] A. Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation operators for ada.

[PM87] R. E. Prather and J. P. Myers, Jr. The path prefix software testing strategy. *IEEE Transactions on Software Engineering*, 13(7) :761–766, July 1987.

[PO01] A. Pretschner and H. Otzbeyer. Model based testing with constraint logic programming : First results and challenges, 2001.

[PP94] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In Thomas Ostrand, editor, *Proceedings of the 1994 Internation Symposium on Software Testing and Analysis (ISSTA)*, pages 58–65. Special issue, ACM SIGSOFT Software Engineering Notes, 1994.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *International Conference on Software Engineering*, pages 105–118, 1992.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, July 2004.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4) :367–375, 1985.

[SD01] N. Tran Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *ASE '01 : Proceedings of the 16$_t$h IEEE international conference on automated software*, pages 13–21, Washington, DC, USA, 2001. IEEE Computer Society.

[SMA05] K. Sen, D. Marinov, and G. Agha. Cute : A concolic unit testing engine for c. In *the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, Lisbon, Portugal, September 2005.

[Spi92] A. Spillner. Control flow and data flow oriented integration test methods. *Softw. Test., Verif. Reliab.*, 2(2) :83–98, 1992.

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3) :229–269, 1997.

[TBJ06] Y. Le Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng.*, 32(8) :571–586, 2006.

[TFW91] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Softw. Test., Verif. Reliab.*, 1(2) :5–25, 1991.

[WF89] D. R. Wallace and R. U. Fujii. Verification and validation : Techniques to assure reliability. *IEEE Software*, 6(3) :8–9, 1989.

[WMM03] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of structural tests for C functions. In *Proc. ICSSEA 2003*, Paris, October 2003.

[WMM04a] N. Williams, B. Marre, and P. Mouy. Interleaving static and dynamic analyses to generate path tests for C functions. In *Proc. SV04*, pages 141–150, Paris, France, December 2004.

[WMM04b] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *Proc. ASE 2004*, pages 290–293, Linz, Austria, September 2004.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *Proc. EDCC 2005*, pages 281–292, Budapest, Hungary, April 2005.

[WNJ97] M. Wallace, S. Novello, and J.Schimpf. *ECLiPSe : A platform for Constraint Logic Programming.* IC-Parc, Imperial College, London, August 1997.

[Won93] W. E. Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, 1993.

[YCA99] N. Yevtushenko, A. R. Cavalli, and R. Anido. Test suite minimization for embedded nondeterministic finite state machines. In *IWTCS*, pages 237–250, 1999.