Université Joseph Fourier - Grenoble I
UFR Informatique et Mathématiques Appliquées

THESE

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

Discipline : Informatique

présentée et soutenue publiquement

par

Karim-Cyril Griche

le 11 juillet 2005

Génération Automatique de bouchons pour le Test Structurel basée sur l'analyse du flot de contrôle

Composition du Jury :

Pierre-Yves Cunin	**Président**
Pascale Le Gall	**Rapporteurs**
Bruno Legeard	
F. Ouabdesselam	**Examinateurs**
I. Parissis	

Remerciements

Je tiens à remercier toute l'équipe VASCO du laboratoire LSR au sein duquel j'ai effectué le travail de recherche présenté dans cette thèse.

Je tiens également à remercier :
– M. Pierre-Yes Cunin pour m'avoir fait l'honneur de présider le jury.
– Mme. Pascale Le Gall et M. Bruno Legeard pour avoir accepté de juger ce travail.

Je tiens tout particulièrement à remercier M. Farid Ouabdesselam et M. Ioannis Parissis, mes deux directeurs de thèse, pour le soutien logistique et moral qu'ils m'ont prodigué tout au long de mon travail et, tout spécialement dans les moments difficiles, de doutes.

Je remercie également :
– L'ensemble de l'équipe administrative du laboratoire LSR, Pascale, Martine, Liliane et Solange qui m'ont guidé dans les méandres de l'administration.
– Francis, Pierre, Olivier, Rémy, Tanguy, Aline, Christophe et Jérôme pour leur présence et leur amitié.
– Ma famille pour son soutien sans faille et ma femme qui a su me montrer que c'était possible.

Finalement, je remercie Abdesselam Lakehal et Corinne Seroze pour leur aide dans la réalisation de ce travail ainsi que toute l'équipe enseignante de l'UFRSFA de Chambéry pour leur acceuil et leur confiance pendant l'année d'ATER que j'ai passée en leur compagnie.

Table des matières

1 Introduction **1**
1.1 Test structurel unitaire et bouchons 2
1.1.1 Méthodes de génération des données de test 2
1.1.2 Les bouchons 3
1.1.3 Liens avec le test d'intégration 3
1.2 Problématique 4
1.3 Contributions 7
1.4 Organisation du document 8

I Contexte général, problématique et proposition **11**

2 Introduction au test, au test unitaire et au test d'intégration **13**
2.1 Types de test 13
2.1.1 Le test dans un cycle de vie 13
2.1.2 Méthodes de test 14
Définitions 15
Le test fonctionnel 18
Le test structurel 18
2.1.3 Le test unitaire et le test d'intégration 21
Le test unitaire 21
Le test d'intégration 21
2.2 Méthodes de génération de données pour le test structurel 24

3 Etude du problème **27**
3.1 Appels de fonctions et bouchons 27
3.1.1 De la nécessité de traiter les appels de fonctions lors du test unitaire 27
3.1.2 Inconvénients des bouchons simples 28
3.2 Analyse de la construction d'un bouchon et techniques associées 29
3.2.1 Analyse préliminaire du processus de création d'un bouchon . . . 29
Notion d'environnement 29
Différentes configurations de "bouchonnage" d'un agrégat 31
Construction d'un bouchon à partir du corps d'une fonction . . . 33
3.2.2 Techniques de transformations et d'approximations de code 34
Techniques de transformations : Slicing et Evaluation partielle . . 34
Technique d'approximation : l'Interprétation Abstraite 37

4 Un bouchon comme une hiérarchie d'approximations **41**
4.1 Approche retenue . . . 41
4.1.1 Synthèse des besoins . . . 41
4.1.2 Complexité de la génération d'une donnée de test . . . 44
4.1.3 Etapes de la création des bouchons . . . 45
4.1.4 Ordre de parcours du graphe d'appel pour la création des bouchons 46
4.2 Approximations et modèle . . . 47
4.2.1 Introduction . . . 47
4.2.2 Approximations . . . 49
4.2.3 Modèle . . . 53
Choix des arcs et critère d'arrêt . . . 53
Complexité des approximations . . . 55
Construction de la hiérarchie . . . 56
4.3 Bouchons . . . 57
4.3.1 Contexte d'une variable . . . 57
4.3.2 Environnement . . . 60
Contexte d'appel . . . 61
Objectifs de génération . . . 62
4.3.3 Deux types de bouchons . . . 65
Filtrage par le contexte d'appel . . . 65
Sélection par les objectifs de génération . . . 67
4.4 Fenêtre d'utilisation des bouchons réalistes . . . 68
4.4.1 Les combinaisons d'approximations . . . 68
4.4.2 Limiter les combinaisons : une fenêtre d'utilisation . . . 69
4.5 Déterminer l'ordre de parcours du graphe d'appel pour la construction des bouchons . . . 70
4.5.1 Ordre d'intégration sans cycle . . . 71
4.5.2 Ordre d'intégration avec cycle . . . 71
4.6 Conclusion . . . 73

5 Illustration sur un exemple **75**
5.1 Présentation de l'exemple . . . 75
5.2 Construction des modèles . . . 80
5.2.1 Une approximation . . . 81
5.2.2 Les modèles d'entités . . . 83
5.3 Constructions des bouchons . . . 87
5.3.1 Calcul de l'environnement . . . 87
5.3.2 Le contexte d'appel . . . 87
5.3.3 Les objectifs de génération . . . 89
5.4 Construction des bouchons dans un environnement . . . 90
5.4.1 Filtrage par le contexte d'appel . . . 91
5.4.2 Sélection pour les objectifs de génération . . . 92

II Application de la méthode sur l'outil Inka 95

6 La programmation logique avec contraintes et Inka 97
6.1 Introduction à la programmation logique avec contraintes 97
6.1.1 Quelques notions de base de ProLog 98
6.1.2 Introduction des contraintes dans la programmation logique . . . 100
6.2 L'approche Inka : la programmation logique avec contraintes pour le test structurel 101
6.2.1 Fonctionnement général 101
6.2.2 Fonctionnement détaillé d'Inka 102
La forme SSA 102
Génération de contraintes pour un langage if-while 104
Fonctionnement des opérateurs de contrôle 105
Définition et appels de fonctions dans Inka 106
6.2.3 Génération d'une donnée de test 108
Etape de filtrage et d'énumération 108
Interprétation des résultats de l'exécution d'Inka pour générer une donnée de test 110
6.2.4 Bénéfice et limites de l'approche 110

7 Implantation à l'aide de l'outil Inka 113
7.1 Choix de représentation 113
7.1.1 Graphe d'appel et ordre 113
7.1.2 Graphe de flot de contrôle pondéré 114
7.1.3 Arcs et imbrication 115
Représentation des arcs 115
Imbrication des arcs 116
7.2 Modèles et bouchons 116
7.2.1 Construction des bouchons 116
Construction d'un modèle 117
Environnement 118
7.2.2 Autre modèle envisagé : la limitation de l'analyse en profondeur du graphe d'appel 121
7.3 Utilisation des bouchons dans Inka 122
7.3.1 Opérateur de gestion des appels de fontions 122
7.4 Expérimentation 124

8 Conclusion et perspectives 127
8.1 Bilan du travail 127
8.2 Approfondir la validation 128
8.3 Perspectives 129

Chapitre 1

Introduction

Durant le développement et la maintenance d'un logiciel, les activités de validation occupent une place très importante aussi bien du point de vue du coût que pour l'obtention de la qualité. Cette qualité porte sur de multiples aspects du logiciel : la fonction réalisée, les performances (occupation mémoire, temps d'exécution), des propriétés de sécurités, de suretés, de fonctionnement, etc.

Nous ne nous intéressons ici qu'au fait que la fonction rendue est conforme aux attentes de l'utilisateur. On peut démontrer cette conformité par une preuve formelle quand on dispose d'une spécification formelle des attentes et que le langage de programmation est doté d'une sémantique formelle. Cette méthode est ardue et seulement partiellement automatisable. L'autre méthode consiste à obtenir une confiance relative dans le logiciel en recherchant des erreurs de fonctionnement lors d'une exécution contrôlée. Les erreurs sont soit recherchées explicitement (existence de modèles de fautes et nécessité d'un environnement d'exécution favorable à la détection d'erreurs), soit implicitement en cherchant à satisfaire un critère donné. C'est *la validation du logiciel par le test*. Le test d'un logiciel est généralement découpé en plusieurs étapes :

- Le test unitaire où on s'assure que chaque entité élémentaire (en général la plus petite partie compilable séparément) du logiciel fonctionne correctement indépendament de l'exactitude des autres entités.
- Le test d'intégration dans lequel on s'assure que les composants du logiciel fonctionnent correctement entre eux.
- Le test système où on teste les interactions possibles du logiciel complet avec un utilisateur, dans un environnement simulé.
- Le test "grandeur nature" (ou d'acceptation) où l'application est réellement installée sur son site d'utilisation.

Chacune de ces étapes peut être réalisée de différentes manières. Elles reposent cependant toutes sur l'utilisation de cas de test et d'un oracle. On appelle *cas de test* le couple formée par une entrée, *la donnée de test*, et la sortie attendue pour cette entrée. Cette sortie est produite par l'oracle qui peut être manuel ou automatique. *L'oracle* est construit à partir d'une spécification explicite du logiciel et des attentes implicites de l'utilisateur.

Il existe deux grandes catégories de techniques mises en œuvre dans toutes les phases de test.

- *Le test fonctionnel (ou "test en boîte noire")*, dans lequel le logiciel est interrogé

comme une boîte dont la réalisation interne est cachée. Les données de test sont engendrées uniquement à partir des spécifications explicites, au minimum le domaine d'entrée du logiciel. Ce type de test n'entre pas dans les thèmes de cette thèse.

– *Le test structurel (ou "test en boîte blanche")*, où la structure du logiciel est connue mais le comportement que le logiciel doit adopter n'est pas la première préoccupation. Les données de test sont sélectionnées pour satisfaire *un critère de test* donné. La satisfaction d'un tel critère vise plus à obtenir un niveau de confiance dans la qualité du logiciel qu'à révéler directement des erreurs de fonctionnement. Chaque critère repose majoritairement sur la *couverture* d'une classe de *composants* du logiciel. Un composant est un élément d'une représentation du logiciel. Si on s'intéresse par exemple aux programmes, un composant peut être une branche ou un chemin du graphe de flot de contrôle ou un couple définition-utilisation du graphe de flot de données. Si on s'intéresse à une spécification à base d'automates, un composant peut être une transition ou un état. La couverture d'un composant nécessite l'exécution du logiciel ou d'une de ses représentations ; elle est effective si le composant considéré a été sollicité. Par exemple, le critère *toutes les instructions* pour un programme a pour but de produire un ensemble de données de test qui permettent de couvrir toutes les instructions de ce programme. On peut fixer un seuil de couverture inférieur à 100% : par exemple, couvrir 80% des transitions d'un automate. Bien que le test structurel ne cherche pas spécifiquement à détecter des erreurs dans un logiciel, on admet que plus le taux de couverture obtenu lors de cette phase de test est élevé, moins l'entité testée est suceptible de contenir d'erreurs et donc plus elle est digne de confiance. Cette affirmation n'a de valeur que parce que la pratique dit également qu'il faut de toute manière pratiquer le test fonctionnel et que les deux formes de test peuvent être spécialisées dans leurs usages pour révéler des erreurs de nature différentes.

1.1 Test structurel unitaire et bouchons

Le test structurel est majoritairement mis en œuvre lors du test unitaire ; il est effectué pour une entité élémentaire du logiciel et pour un critère de test donné. Les méthodes de test les plus largement utilisées dans l'industrie impliquent l'utilisation de mesures de couverture des entités du logiciel.

1.1.1 Méthodes de génération des données de test

On peut distinguer deux manières de procéder lors du test structurel unitaire.

On peut mesurer un taux de couverture obtenu *a posteriori* après exécution pour un ensemble de données de test. C'est le cas quand les données sont engendrées aléatoirement ou lorsqu'on réutilise des données de test déjà produites, par exemple lors d'une phase de test fonctionnel de l'entité.

Lorsque le taux de couverture résultant n'est pas satisfaisant pour le testeur ou en l'abscence de données disponibles, on utilisera des techniques de génération de données de test *a priori*, en recherchant une donnée de test qui va permettre d'exécuter un composant pré-sélectionné dans l'entité sous test. La génération de telles données est l'une des

activités les plus importantes et les plus coûteuses en temps lors du test structurel unitaire d'un logiciel. En effet, le test aléatoire ne permet que rarement d'atteindre l'objectif fixé par le critère de test.

De nombreuses recherches portent sur la définition et la réalisation d'outils permettant l'automatisation de la production de ces données. Ces recherches se classent généralement en deux catégories : celles fondées sur une analyse statique du code du programme [MS04], les plus courantes, et celles qui sont basées sur une recherche dynamique des données de test par de multiples exécutions du programme [Kor90, Kor96, FK96].

1.1.2 Les bouchons

En phase de test unitaire, chaque entité constituante du logiciel est testée seule : aucune interaction avec d'autres entités du logiciel n'est prise en compte. Ces interactions sont court-circuitées grâce à l'utilisation de *bouchons*. Un bouchon a la même structure syntaxique que l'entité qu'il remplace : procédure, méthode, classe...

L'introduction d'un bouchon a pour but d'éviter la complexification la génération de données. En effet, lorsque l'exécution d'un composant de l'entité sous test dépend d'une interaction avec une autre entité du logiciel, le choix des données de test permettant la couverture de ce composant nécessite l'analyse de cette deuxième entité. Or cette analyse est en général aussi complexe que celle de l'entité sous test.

Un bouchon se présente donc comme une entité très simple dont l'utilisation ne doit pas avoir d'impact sur la difficulté à générer des données de test. Au pire, il fournit à chaque invocation, un même résultat qui appartient au domaine des valeurs de l'entité appelée. Au mieux, il simule le comportement de cette entité dans chacun de ses contextes particuliers d'appel.

Par exemple, lors du test structurel unitaire d'une fonction $f()$ pour le critère "toutes les branches", si on rencontre un appel à une fonction $g()$, cet appel est remplacé par un appel à la fonction $bg()$, un bouchon de $g()$, créé spécifiquement comme sur la figure 1.1.

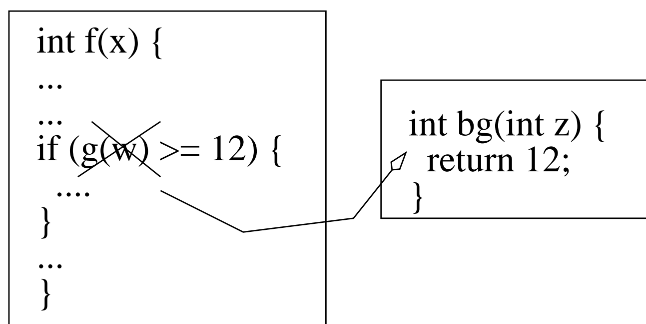

FIG. 1.1 – Un bouchon de g()

Ces bouchons sont généralement construits à la main par le testeur à partir d'une spécification (formelle ou informelle) et souvent incomplète de l'entité bouchonnée.

1.1.3 Liens avec le test d'intégration

Lors de la phase d'intégration, les différentes entités testées indépendamment sont rassemblées, une à une ou sous forme d'agrégats, pour former le logiciel. A chaque ajout,

on effectue un test d'intégration. Par rapport au test unitaire, l'intégration supprime les bouchons au profit des entités originales. A chaque étape de l'intégration l'activité de test reçoit pour mission de produire des données de test qui activent les interactions entre les entités intégrées jusqu'ici afin d'y déceler les erreurs éventuelles.

Les critères de test structurel unitaire sont considérés comme trop fastidieux à mettre en œuvre lors de l'intégration. Lorsqu'on retire les bouchons, la génération de données de test structurel est considérablement complexifiée. On a vu qu'elle nécessite l'analyse de chaque entité appelée dans son contexte avant de pouvoir produire une donnée de test.

Le critère de couverture de la phase d'intégration est souvent une transposition des critères structurels unitaires au graphe d'appel d'un agrégat. On appelle graphe d'appel d'un logiciel le graphe qui représente les interactions entre les entités du logiciel. Une étape du test d'intégration est terminée lorsqu'on obtient un taux de couverture jugé suffisant des arcs du graphe d'appel de l'agrégat sous test.

De même que pour le test unitaire, le test d'un agrégat nécessite de bouchonner les appels vers les entités qui n'ont pas encore été intégrées. Le nombre de bouchons à produire dépend fortement de la politique d'intégration choisie pour le logiciel. Pourtant, quelle que soit cette politique, les bouchons peuvent être soit très nombreux soit compliqués à produire. Leur génération représente une grande part de l'effort de test à fournir.

A notre connaissance, les publications concernant le test d'intégration ne proposent pas d'automatisation des techniques de test. Il s'agit plutôt de politiques d'intégration qui facilitent le test d'intégration "à la main" des agrégats en limitant notamment l'effort de bouchonnage nécessaire à chaque étape.

Certains travaux ont cherché à définir des critères de couverture spécifique à la phase de test d'intégration [MB89]. Malgré tout, l'industrie du logiciel se base encore majoritairement sur l'utilisation de critères de couverture tels qu'ils sont mis en œuvre dans le test structurel unitaire. Pendant le test d'intégration, les agrégats font l'objet de test fonctionnel éventuellement complété par une mesure de couverture du graphe d'appel.

1.2 Problématique

Cette thèse aborde le problème de la création de bouchons qui représentent tout ou partie des comportements possibles des entités qu'ils remplacent. Ce problème est déterminant dans la conservation de la confiance placée dans la qualité de la structure du logiciel à l'issue de la phase de test structurel unitaire. En effet, la confiance dans une entité est représentée par le taux de couverture obtenu à la fin de son test unitaire. Quand on remplace les bouchons par l'entité originale, on ne peut préserver la confiance que si les taux de couverture observés sont maintenus.

Ce problème est tout à fait d'actualité dans l'industrie du logiciel où le test structurel fournit les principales mesures de qualité du logiciel. Même dans le domaine spécifique des systèmes critiques, tel que l'avionique à travers les normes DO178A et B, le test structurel est absolument requis.

Or, on note que les bouchons utilisés dans le test unitaire sont par nature imprécis. Ils sont construits par le testeur à partir d'une spécification, souvent incomplète, de l'entité bouchonnée. L'impact principal de cette imprécision se manifeste par la sélection de

données de test qui peuvent être erronées. Le comportement de l'entité sous test obtenu par exécution de ces données n'est pas celui qui aurait lieu en présence de la fonction appelée et non de son bouchon. De ce fait, certains composants de l'entité sous test (appelante) sont atteints alors qu'ils ne le seraient pas sans l'utilisation du bouchon. Inversement il peut arriver que l'utilisation d'un bouchon en dehors des comportements initialement prévus rende artificiellement des composants de l'entité appelante impossibles à atteindre.

On observe également que le problème est amplifié par la rupture qui existe entre les techniques mises en œuvre respectivement pendant les phases de test unitaire et d'intégration. Lors de la première on s'intéresse surtout au graphe de flot de contrôle (ou de données ou celui de données interprocédurales) ; pendant la seconde la représentation retenue est le graphe d'appel. On ne peut donc pas observer l'évolution des taux de couverture des critères structurels unitaires lors de la phase d'intégration.

Enfin, l'arrivée des technologies orientée objet, où l'intégration d'entités souvent pré existantes (réutilisation) prend un rôle prépondérant dans la création de logiciels, rend plus prégnant le problème de la création de bouchons assurant la préservation du niveau de confiance obtenu en test unitaire.

L'approche présentée dans cette thèse vise à étudier des techniques permettant la génération automatique de données de test structurel unitaire réalistes en présence d'appels d'autres entités du logiciel. Une prise en considération des appels entre entités au niveau du test unitaire doit être réaliste de manière à obtenir un ensemble de données de test qui représente au mieux la couverture de l'entité dans le logiciel mais rester suffisamment simple pour permettre l'utilisation des critères de couvertures structurels unitaires.

Nous appelons *le système sous test* l'agrégat d'entités du logiciel sur lequel nous réalisons le test structurel unitaire.

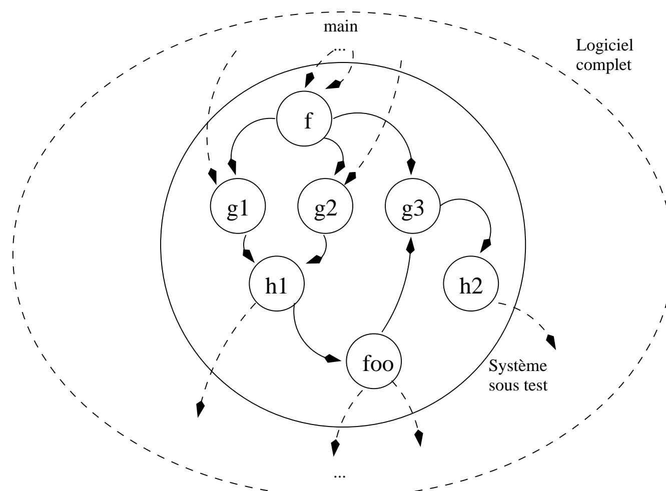

FIG. 1.2 – Un système sous test

Un système sous test est composé d'un ensemble fini d'entités du logiciel. Comparé à l'agrégat du test d'intégration, le système sous test n'évoluera pas au cours du temps, aucune entité ne lui sera ajoutée, son test sera réalisé en une seule étape.

Comme dans le test d'intégration, il existe des entités qui appellent ou qui sont appelées par le système sous test et qui n'en font pas partie. Lors du test unitaire d'un tel système, ces entités sont remplacées par des bouchons ou des lanceurs (entité simplifiée qui appelle une entité du système sous test). L'évaluation de leur conformité ne fait pas partie des objectifs de ce test de la même façon qu'on ne s'assure pas de la conformité des entités externes à un agrégats à un instant donné.

Réaliser la phase de test unitaire d'un ensemble d'entités regroupées au sein d'un système sous test consiste à produire, pour toutes les fonctions de ce système, un ensemble de données de test assurant un pourcentage requis de couverture d'un critère de test structurel unitaire. Durant cette phase de test, la génération de données de test pour une entité précise ne cherche pas à couvrir en même temps toutes les entités appelées. Lorsqu'on cherche à couvrir une fonction f de la figure 1.2 pour le critère de couverture "toutes les branches", on ne se préoccupe pas de couvrir en même temps les branches des fonctions g_1, g_2, g_3 appelées par f.

Ainsi le problème posé par le test d'un agrégat d'entités se situe dans la prise en compte de g_1, g_2, g_3 pour produire des jeux de test réalistes assurant la couverture de f pour un critère donné.

La traitement des appels entre entités lors du test structurel unitaire n'est pas satisfaisant actuellement dans la pratique industrielle ; il ne fait pas plus l'objet, à notre connaissance, de travaux de recherche. Il n'existe pas d'alternative entre le bouchonnage simple et très imparfait des appels et le dépliage complet et systématique du corps des entités appelées en vue de leur analyse. De plus, il n'existe pas d'outil de génération automatique de bouchons autres que triviaux retournant une valeur prédéfinie qui dépend uniquement du type de retour de l'entité originale : 0 pour *int*, *null* pour un structure dynamique ou un pointeur par exemple.

Il est donc nécessaire d'envisager une approche médiane entre ces techniques classiques, c'est à dire proposer une représentation des entités appelées dans leur contexte qui soient suffisamment réaliste pour garantir une représentativité des données de test produites mais assez simple pour envisager un passage à l'échelle de la méthode par le traitement efficace de programmes comportant un grand nombre d'appels de fonctions.

Enfin cette thèse a été effectuée dans un cadre particulier : un projet RNTL de type précompétitif, nommé INKA. Ce projet avait pour but l'industrialisation du prototype du même nom [Got00, GBR00]. INKA est un outil de test structurel unitaire dont la spécificité réside dans l'utilisation de la programmation logique avec contraintes pour produire les données de test permettant d'atteindre un composant donné dans l'entité sous test, ici des fonctions écrites en C.

Dans ce projet, à coté de l'industrialisation réalisée par Thalès Systèmes Aéroportés, les partenaires universitaires ont été chargés d'étudier et de mettre en œuvre différentes extensions de ce prototype. Chaque extension fait l'objet d'un sous-projet propre : extension au traitement des flottants (avec l'équipe "contraintes" de I3S - Université de Nice), et extension aux structures dynamiques (avec l'équipe Techniques à Contraintes du LIFC - Université de Besançon). Le sous-projet auquel nous avons participé s'intitule "extension au test d'intégration".

1.3 Contributions

Cette thèse aborde le problème posé par la génération automatique de données de test structurel en présence d'appels de fonctions. Nous proposons une technique permettant l'analyse d'une représentation réaliste des fonctions appelées au sein du processus de test. Cette technique s'appuie sur la création automatique de *bouchons réalistes* des fonctions appelées du *système sous test*.

L'idée générale de cette thèse part de la constatation suivante : les bouchons classiques sont trop simplistes pour permettre de générer des données représentatives des comportement possibles des fonctions qu'ils remplacent. En conséquence le taux de couverture obtenu pour le critère structurel retenu lors du test unitaire n'est pas représentatif de l'utilisation de la fonction testée au sein du logiciel. Par ailleurs il n'est pas envisageable d'analyser le corps des fonctions appelées originales lors du test structurel unitaire des fonctions appelantes ; le système qui en résulterait étant beaucoup trop complexe pour envisager son traitement.

Nous proposons une alternative : modéliser les fonctions appelées de façon *réaliste pour les besoins du test structurel*. Ces modèles sont réalistes si l'application du processus de génération de données de test à une entité "appelant" un tel modèle fournit la même donnée de test que l'application du même processus à cette entité appelant la fonction originale. Ce modèle est généré *automatiquement* à partir d'une analyse statique du corps de la fonction appelée, comme précisé ci-dessous.

Le modèle d'une fonction est constitué d'un ensemble d'approximations de cette entité. Un bouchon est construit à partir du modèle ; il est composé d'une hiérarchie d'approximations de plus en plus complexes. Chaque approximation représente une classe de comportement de l'entité originale. Pour la génération des données de test, ce bouchonnage en couches a le double avantage de pouvoir simuler l'ensemble des comportements possibles de la fonction originale et d'éviter d'avoir à considérer tous ces comportements en même temps. En effet, les comportements les plus simples à analyser sont placés en tête du bouchon. Ils seront traités en premier lorsqu'un appel vers la fonction bouchonnée sera rencontré. Lorsque les comportements simples ne sont pas suffisants pour générer une donnée nécessaire à atteindre un composant de la fonction appelante - i.e. l'approximation de la fonction appelée ne permet pas de générer une sortie nécessaire à atteindre le composant de la fonction appelante - on augmente le réalisme de l'approximation en avançant dans la hiérarchie. L'approximation suivante simulera plus de comportements de la fonction originale augmentant du même coup les chances de générer une donnée suffisante au prix d'une analyse des appels de fonctions légèrement complexifiée.

Ces bouchons qui sont fidèles aux comportements des fonctions qu'ils remplacent sont appelés *bouchons réalistes*. Leur création est effectuée en plusieurs étapes. Tout d'abord on opère une analyse statique de toutes les entités du système sous test. Cette analyse permet de construire le graphe de flot de contrôle de chaque entité ainsi que le graphe d'appel du sytème sous test.

On s'appuie sur ce graphe d'appel pour déterminer l'ordre de création des bouchons réalistes. Cet ordre est pensé de manière à maximiser, lorsque c'est nécessaire et à chaque étape de la création, la réutilisation des bouchons qui ont été créés aux étapes précédentes.

Les approximations d'une entité sont construites par découpage de son graphe de flot

de contrôle. Par ce procédé, on crée un ensemble de sous-graphes représentant chacun une classe de comportements de l'entité originale. Une pondération du graphe de flot de contrôle nous permet de donner un poids aux sous-graphes.

Enfin on construit un bouchon pour chaque appel à l'entité en intégrant, dans l'ordre de leur poids, les approximations extraites de l'ensemble et qui correspondent à l'environnement particulier de chaque appel. En effet, un appel de fonction détermine un contexte d'utilisation du bouchon et les approximations n'ont pas forcément de sens dans tous les contextes d'utilisation. Nous avons définie cet environnemnt pour des variables de types scalaires ; il se découpe en deux parties : le *contexte d'appel* et les *objectifs de génération*. Par exemple, il n'est pas nécessaire d'intégrer, dans un bouchon, le comportement de la fonction *Valeur_Absolue()* appelée avec des entiers négatifs lorsque l'environnement de son appel implique l'utilisation d'entiers supérieur à 10.

En résumé, l'utilisation des bouchons réalistes à la place des bouchons simples va permettre une génération de données de test plus pertinentes en rendant l'analyse des appels de fonctions moins complexe que celle qu'aurait entrainé le dépliage complet du corps des fonctions.

Cette proposition a été implantée en utilisant les capacités du prototype INKA. Nous avons développé un prototype de création automatique des modèles des fonctions appelées pour un agrégat de fonction. Nous sommes également en mesure de calculer le contexte d'appel d'une fonction en un point particulier de la fonction appelante. Ce contexte nous permet d'invalider certaines approximations du modèle de la fonction appelée.

1.4 Organisation du document

Ce document s'articule autour de deux grandes parties :

- La partie I est composée de quatre chapitres. Tout d'abord, dans le chapitre 2.1, nous présentons des méthodes de test en général et plus particulièrement des méthodes employées pour parvenir à la génération automatique de données de test structurel. Ce chapitre présente également les différentes politiques d'intégration des entités au sein d'un logiciel.
  Dans le chapitre 3, nous présentons la problématique de cette thèse, une étude préliminaire des problèmes qu'elle pose et une étude bibliographique spécifique à notre analyse de la problématique.
  Notre proposition est présentée au chapitre 4. Nous commençons par présenter une synthèse des besoins que nous avons identifiés au chapitre 3. Les sections suivantes sont consacrées aux différentes étapes successives de la création automatique des bouchons : extraction d'un modèle des fonctions, création des environnements d'appel et finalement, construction des bouchons spécifiques à ces environnements.
  Enfin le chapitre 5 illustre, sur un exemple simple, toutes les étapes de création de ces bouchons.
- La partie II est consacrée à l'implantation de la proposition. Cette partie s'articule autour de trois chapitres. Le chapitre 6 est réservé à une présentation rapide de la programmation logique avec contraintes puis à l'utilisation généralement faite de ce type de programmation dans le test.
  Le chapitre 6.2 présente le fonctionnement détaillé du prototype INKA. C'est au

dessus de ce dernier que nous avons implanter la création automatique de bouchons pour le test structurel. Les détails de cette implantation et les premiers résultats qui peuvent en être tiré font l'objet du dernier chapitre de cette partie.

Première partie

Contexte général, problématique et proposition

Chapitre 2

Introduction au test, au test unitaire et au test d'intégration

2.1 Types de test

Le test d'un logiciel est effectué pour s'assurer empiriquement de son bon fonctionnement. Pour y parvenir, une des méthodes consiste à solliciter le logiciel avec des entrée de manière à y détecter le plus d'erreurs possible. Le logiciel est exécuté avec une donnée de test, la sortie ou le comportement du logiciel est alors analysé afin de vérifier si il est incorrect. La vérification du comportement d'un logiciel est réalisée par un oracle qui donnera un verdict pour chaque test. Cet oracle peut prendre différentes formes telles que des spécifications formelles ou informelles sous forme de cahier des charges ou encore être réalisé à la main grâce à l'expertise du testeur. Pour chaque donnée de test, l'oracle la compare les informations issues du cas de test, c'est à dire le couple entrée fournie sortie obtenue à une spécification considérée correcte du logiciel. Il fournit alors un verdict pour le cas de test, une erreur est détectée dans le logiciel lorsque la sortie effective ne correspond pas à celle attendu par les spécifications.

2.1.1 Le test dans un cycle de vie

La vie d'un logiciel commence dès l'élaboration sur papier de ses futures fonctionnalités. Un logiciel ne meurt pas, il continue à vivre tant qu'il est utilisé. Pour le concepteur l'existence d'un logiciel n'est pas cantonné à son simple développement. Sa maintenance tout au long de son utilisation ainsi que l'ajout de fonctionnalités demandées par les utilisateurs font partie de la vie d'un logiciel.

Le développement d'un logiciel est effectué en plusieurs étapes qui sont partie intégrante de son cycle de vie. Ces étapes sont de natures différentes, on y trouve la spécification du logiciel, son développement ou encore son test. Nous nous intéressons particulièrement aux étapes de test dans le cycle de vie d'un logiciel. Elles sont généralement effectuées au plus tôt dans le développement du logiciel de manière à minimiser l'impact d'éventuelles erreurs pour son coût de production. Elles perdurent tout au long de sa vie lorsque de nouvelles fonctionnalités sont ajoutées.

Dans le cycle de développement d'un logiciel on trouve plusieurs étapes de test successives :

– Tout d'abord on effectue des tests sur les plus petites entités compilables séparement du logiciel. Les tests de chaque entité sont indépendants les uns des autres. C'est l'étape de *test unitaire.*
– Ces entités sont alors assemblées. Les agrégats d'entités font l'objet de nouveaux tests qui visent à mettre aux jours d'éventuels défauts dans la communication des entités au seins de l'agrégats. C'est la phase de *test d'intégration.*
– Lorsque toutes les entités du logiciel sont intégrées, on cherche à tester les fonctionnalités du logiciel complet pour s'assurer qu'elles correspondent bien au besoin de l'utilisateur final. C'est la phase de *test de validation.*
– Enfin si, au cours du temps, on complète ces fonctionnalités par de nouvelles, il faudra s'assurer de leur bon fonctionnement mais également que cet ajout ne détériore pas les anciennes. C'est la phase de *test de non régression.*

Ces grandes phases de test sont présentes dans de nombreux cycles de vie. Le plus connu d'entre eux est certainement le cycle de vie en "V" de la figure 2.1 où la branche descendante du cycle correspond aux phases de développement et où la branche montante correspond aux tests effectués à chaque étape du développement.

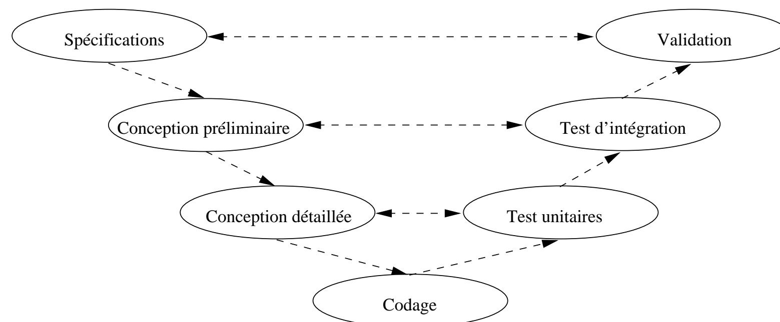

FIG. 2.1 – Un cycle de vie en V

2.1.2 Méthodes de test

Il existe dans la littérature de très nombreuses méthodes de test. Elles peuvent être grossièrement classées en deux catégories : les méthodes de test fonctionnel et les méthodes de test structurel. Bien que certaines de ces méthodes se situent à la frontière entre les deux classes, on peut généralement les distinguer par leur vue du logiciel comme entité de test.

Dans une approche fonctionnelle, le logiciel ou ses entités sont considérés comme atomiques. Une entité a une structure de fonctionnement cachée, on ne s'intéresse qu'à ses fonctions pendant le test.

A contrario dans l'approche structurelle, le fonctionnement interne du logiciel est connu. Le logiciel peut être considéré comme un ensemble de chemins d'exécution répondant chacun à un comportement attendu par l'utilisateur. Pendant le test on s'assurera de la qualité de la structure interne du logiciel.

Définitions

Définition 2.1 *Une condition est une expression à valeur booléenne ne comportant aucun opérateur logique binaire (et, ou, xor, ...). Par exemple,* $A > 1$ *est une condition.*

Définition 2.2 *Une décision est une expression à valeur booléenne construite à partir de conditions et d'opérateurs logiques binaires. Par exemple,* $(A > 1)et(B = 0)ou(C <= 3)$ *est une décision.*

Définition 2.3 *Les instruction élémentaires sont les instructions d'affectation ainsi que celles utilisées pour effectuer les entrées/sorties.*

Définition 2.4 *Un bloc d'instructions est une suite d'instructions élémentaires telle que, si le flot d'exécution arrive à la première instruction du bloc alors toutes ses instructions seront exécutées dans leur ordre d'apparition au sein du bloc.*

Définition 2.5 *Un appel de fonction est un point dans le programme où les paramètres d'entrées de la fonction appellée reçoivent une valeur et où le contrôle est transmis du programme à la fonction appelée.*

Remarquons que, dans ce document, aucune différenciation n'est faite entre l'appel d'une fonction ou d'une procédure. Les termes seront employés indifféremment dans la suite.

Définition 2.6 *Une instruction de contrôle (si...alors...sinon, boucles ou switch) oriente le flot d'exécution du programme en fonction de la valeur de la décision sur laquelle elle repose.*

Par exemple : si $(A > 1)$ *et* $(B = 0)$ *ou* $(C <= 3)$ *alors* $Y = 3$ *sinon* $Y = 4$ *ou* $Tant\ que\ (A > 1\ et\ B = 0)\ faire\ A = A - 2\ fin.$

On peut noter que l'instruction de contrôle switch est un cas particulier car elle peut reposer sur plusieurs décisions. Par exemple : switch(A) case 1 *: ...., case* 2 *: ...etc repose sur autant de décisions qu'il y a de choix dans le switch.*

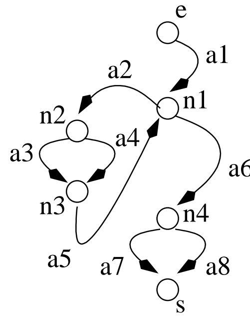

FIG. 2.2 – Le graphe de flot de contrôle de la procédure tordu

```
void tordu(int t[], int borne) {
        int i,j ;

        i = j ;
        i = 0 ;
        while (i < borne) {
                if ((t[i] % 2) == 0)
                        printf("%d\n", t[i]) ;
                scanf("%d", &j) ;
                t[i] = t[j] + 1 ;
        }
        j=2*j ;
        if (j>0) {
                i = 0 ;
                printf("possible\n") ;
        } else {
                printf("impossible\n") ;
        }
}
```

Définition 2.7 *Le graphe de flot de contrôle (GFC) d'un fonction ou d'une procédure est un graphe orienté* $G = (N, A, e, s)$ *où :*

- N *est l'ensemble des nœuds du graphe,*
- A *est l'ensemble de ses arcs,*
- $e \in N$ *est son unique nœud d'entrée,*
- $s \in N$ *est son unique nœud de sortie.*

De plus, chaque nœud $n \in N$ *possède :*

- *un degré d'entrée, noté* $In(n)$*, déterminé par le nombre d'arcs dont les extrémités parviennent à ce nœud,*
- *un degré de sortie, noté* $Out(n)$*, déterminé par le nombre d'arcs qui ont leurs origines à ce nœud.*

Enfin, à chaque nœud $n \in N$ *appartient à une des catégories suivantes :*

- *Nœuds de décision (type d) qui sont associés aux décisions des instructions de contrôle du langage* ($Out(n) > 1$)*.*

- *Nœuds de jonction (type j) qui représentent les points de rencontre de plusieurs arcs ($In(n) > 1$).*
- *Nœuds d'appel (type a) qui représentent les appels de fonctions ou de procédures ($Out(n) = 1$).*

Chaque arc $a \in A$ appartient à une des deux catégories :

- *Arc de bloc (type b) qui représente un bloc d'instructions de la fonction.*
- *Arc de retour (type r) qui représente l'affectation des variables définies par l'exécution de la fonction ou de la procédure appelée au nœud de type appel à l'origine de l'arc. Un arc de retour peut correspondre à plusieurs situations : affectation explicite de la valeur d'une fonction à une variable, affectations implicites de valeurs aux paramètres résultats d'une procédure, affectations implicites de valeurs à des variables globales présentes dans la fonction appelée.*
  Cet arc peut être en même temps un arc bloc pour prendre en compte toutes les instructions qui suivent immédiatement l'appel à la fonction.

On peut noter que pour faciliter la lecture des graphes de flot de contrôle, un nœud de jonction pourra également être du type décision ou appel.

Par exemple, le graphe de flot de contrôle de la procédure "tordu" de l'exemple 2.1.2 est représenté sur la figure 2.2. Il est noté

$$G_{tordu} = (\{e, n_1, n_2, n_3, n_4, s\}, \{a_1, a_2, a_3, a4, a_5, a_6, a_7, a_8\}, e, s)$$

où les nœuds n_1, n_2, n_4 sont des nœuds de décisions et les nœuds n_1, n_3, s sont des nœuds de jonction. Il n'y a pas de nœuds d'appel dans ce GFC ; tous ses arcs sont des arcs de blocs.

Définition 2.8 *Un chemin c dans $G = (N, A, e, s)$ est une suite d'arcs a_i de A issue de e et allant jusqu'à s telle que l'extrémité de l'arc a_i est l'origine de l'arc a_{i+1}.*

Définition 2.9 *Un chemin d'exécution d'une fonction est un chemin de son graphe de flot de contrôle tel qu'il existe une entrée de la fonction qui provoque son cheminement par le flot d'exécution.*

Par exemple, $a_1, a_2, a_4, a_5, a_6, a_7$ est un chemin du GFC de la procédure tordu. Cette procédure n'admet aucun chemin exécutable.

Définition 2.10 *Une variable est définie par une affectation explicite quand sa valeur est déterminée par une instruction élémentaire. Elle est définie par une affectation implicite quand sa valeur est déterminée par un appel de fonction.*

Une variable est utilisée (ou référencée) lorsqu'elle sa valeur est utilisée dans une instruction élémentaire ou dans une condition.

Par exemple l'instruction élémentaire $x = y + 1$ définit la variable x et utilise la variable y.

Le test fonctionnel

Dans cette approche du test, le logiciel ou une de ses entités est vu comme une boîte noire. La seule information dont un testeur dispose est la manière dont cette boîte réagit aux stimuli externes. On étudie le comportement fonctionnel du programme. Le test fonctionnel est utilisé pour vérifier la conformité des réactions du logiciel avec les attentes des utilisateurs (sa spécification). En somme, on s'assure que le logiciel fait bien ce qu'on lui demande de faire.

Le test fonctionnel se décompose en trois phases : tout d'abord la sélection des données de test pour le logiciel, puis l'exécution de ces données, enfin le verdict : le logiciel a-t-il réagi comme il le devait ? Ces étapes forment la base des techniques de test fonctionnel.

Il existe de nombreuses techniques qui se différencient essentiellement par la manière dont on sélectionne les données de test. On sait qu'en pratique le test exhaustif d'un logiciel n'est pas envisageable. Différentes études ont porté sur la formalisation de critères de sélection des données de test allant de la plus simple : le test aléatoire, à des techniques plus évoluées telles que le test de partition, où les entrées du logiciel sont regroupées au sein de classes d'équivalence (les partitions) pour ne sélectionner qu'une donnée aléatoire dans chaque partition, ou encore le test aux limites, considéré comme une des méthodes les plus efficaces, où on se base sur les bornes des domaines de définitions des variables pour proposer des données de test.

Le test structurel

Lorsqu'on parle de test struturel, on entend test structurel unitaire, à notre connaissance il n'en existe pas d'autre type.

On appelle :

Définition 2.11 *Les composants d'une fonction, les éléments* arc, nœud, chemin, condition *qu'on peut trouver dans le graphe de cette fonction ou encore ceux qui peuvent en être dérivé tel que* les conditions/décisions, conditions/décisions modifiées.

Définition 2.12 *La couverture d'un composant, posséder une donnée de test dont l'exécution garantie l'activation de ce composant dans la fonction.*

Définition 2.13 *le critère structurel, le type de composant d'une fonction dont le test structurel doit assurer la couverture.*

Définition 2.14 *Niveau de couverture, le pourcentage des composants du critère structurel couverts par l'ensemble des données de test produites.*

Le test structurel unitaire consiste à produire un ensemble de données de test assurant un niveau de couverture déterminé pour un critère structurel donné d'une entité du logiciel.

Le test structurel unitaire ne se substitue pas au test fonctionnel d'une entité. Bien que les mesures de couvertures structurelles ne cherchent aucunement à mettre au jour des erreurs dans la structure interne des entités, il est admis qu'elles ajoutent un niveau de confiance quant à la qualité du code qui les composent.

Suivant le critère structurel que le testeur a retenu pour le test unitaire, les données de test générées vont permettrent de s'assurer de l'absence de certains types d'erreurs dans le code des entités.

Les critères de test structurel sont classés en catégorie suivant qu'ils portent sur le graphe de flot de contrôle, le graphe de flot de données, le graphe d'appel, etc. Chacun de ces critères a des avantages, lié aux types d'erreurs dont ils garantissent l'abscence, et des inconvénients, généralement en rapport avec la difficulté à générer des données assurant le niveau de couverture souhaité.

Il existe de nombreux critères structurels, dans notre travail nous nous sommes essentiellement intéressé aux critères portant sur le graphe de flot de contrôle et en particulier au critère de couverture des arcs (ou des décisions).

Nous donnons cependant ici un rapide aperçu des critères structurels usuels, de leurs avantages et de leurs inconvénients :

- Critères courants concernant le graphe de flot de contrôle :
  - *Couverture des instructions élémentaires.* On produit assez de données de test pour que toutes les instructions d'une entité soient exécutées au moins une fois. Cette mesure est très souvent utilisée. Un des arguments généralement avancé pour expliquer cette utilisation est que les erreurs de programmation sont distribuées équitablement entre toutes les instructions du graphe de flot de contrôle. Par conséquent un taux élevé de couverture de ces instructions donne une bonne approximation de la qualité du code du composant.
    En contrepartie, cette mesure ne permet pas de détecter efficacement des problèmes liés au flot de contrôle de la fonction. Par exemple, si une fonction possède une seule instruction de branchement if et que les opérations de cette fonction ne sont pas également réparties dans les arcs (par exemple une opération dans la branche then et les 99 autres dans la branche else), on peut obtenir des résultats variant entre 1 et 99% de couverture suivant les données de test produites.
  - *Couverture des décisions (ou des arcs).* Cette mesure donne la couverture des arcs issus des instructions de contrôle de la fonction. On va produire des données de test assurant que chaque décision de la fonction a été évaluée au moins une fois à $vrai$ et une fois à $faux$. Cette mesure inclut généralement la mesure de couverture des instructions ; celles-ci étant réparties sur les arcs de la fonction.
  - *Couverture des conditions.* Cette mesure donne la couverture de chaque condition des décisions de la fonction. On va produire des données de test de manière à évaluer au moins une fois à $vrai$ et une fois à $faux$ toutes les conditions des décisions. Cette mesure est plus forte que la couverture des décisions, on teste les combinaisons de valeurs des variables qui interviennent dans les décisions. Pourtant, une couverture complète des conditions ne garantit pas la couverture des décisions. Par exemple, les conditions de l'instruction de contrôle Si $(A\ et\ B)$ $alors$ ... peuvent être couverte par deux données de test assurant que $A = faux, B = vrai$ et $A = vrai, B = faux$, pour autant la décision $A\&B$ ne prendra jamais la valeur $vrai$.
  - *Couverture des conditions/décisions.* C'est une mesure hybride entre la couverture des conditions et la couverture des décisions. Elle combine la précision de la couverture des conditions tout en assurant la couverturee des décisions. Ce critère ne parvient pas toujours à couvrir toutes les combinaisons de valeurs des condi-

tions, les opérateurs logiques binaires *et*, *ou* peuvent l'en empêcher. Par exemple, si une condition d'un *et* est fausse, les valeurs des autres conditions de la décision ne seront pas évaluées. De même, lorsqu'une condition d'un *ou* est vraie.

  - *Couverture des conditions multiples.* Cette mesure donne la couverture de toutes les combinaisons de valeurs des conditions dans la décisions. Les données de test requises pour la couverture maximale des conditions multiples est donnée par la table de véritée des opérateurs logiques binaires de la décision.

    Cette mesure est une extension de la couverture des conditions. Elle est difficile à mettre en pratique car il est fastidieux de déterminer le nombre de cas de test minimal à produire pour cette mesure. De plus le nombre de cas de test requis peut varier énormement entre deux décisions qui ont le même nombre d'opérateurs et d'opérandes.
  - *Couverture des conditions/décisions modifiées (MC/DC).* Pour obtenir une couverture maximale des MCDC, on doit produire assez de cas de test pour s'assurer que chaque condition peut affecter le résultat de sa décision englobante. Elle a été originalement introduite chez Boeing et est maintenant demandée pour tous les logiciels avioniques.
  - *Couverture des chemins.* Cette mesure nécessite de produire assez de données de test pour exécuter chaque chemin de la fonction. Les boucles peuvent induire un nombre infini de chemins dans la fonction, on mesure généralement un sous ensemble de tous ces chemins définis par un nombre d'itération dans les boucles. Par exemple, on passera zéro ou une fois dans le corps d'une instruction *while*.

    Cette mesure requièrt un test minitieux des fonctions. Elle est soumise à deux inconvénients majeurs : (a) le nombre de chemins d'une fonction augmente de façon combinatoire à son nombre de décisions, (b) un grand nombre de ces chemins ne sont pas des chemins d'exécution.
- D'autres critères courants concernent le graphe de flot de donnée, tels que les critères de couvertures des définitions, des utilisations ou des définitions/utilisations. Ces critères peuvent être vus comme des variations du critère de couverture des chemins dans lequel on ne considère que les sous-chemins de la définition d'une variable à ses différentes utilisations ultérieures.

  L'avantage majeur de ce type de critères par rapport au critère de couverture des chemins portant uniquement sur le flot de contrôle est qu'on ne s'intéresse qu'aux chemins qui ont un impact direct sur la manière dont le programme va traiter ses données. Cette mesure est complexe à mettre en œuvre, de plus elle n'inclut pas le critère de couverture des décisions.
- Autres critères : il existe de nombreux autres critères plus complexes qui sont, pour la plupart, issu des critères présentés précédement comme par exemple les critères de couvertures des portions linéaires de code sans sauts.

Dans la suite de cette thèse nous ne nous intéresserons uniquement aux critères basés sur le graphe de flot de contrôle et, sauf mention contraire, du critère de couverture des arcs.

2.1.3 Le test unitaire et le test d'intégration

Le test unitaire

Le test unitaire d'une entité peut être effectué dès que son implantation est terminée. L'entité en elle même peut prendre différentes formes suivant le type de programmation utilisée. Elle sera une procédure ou une fonction dans le cas de la programmation dans un langage itératif usuel ou par exemple une méthode ou une classe pour la programmation orientée objet. Dans tous les cas l'entité à tester possède des entrées et des sorties, de plus elle réalise une opération particulière et identifiée au sein du logiciel.

Le test unitaire peut être effectué en utilisant indifféremment toute la palette des techniques de test fonctionnel et/ou structurel. En fonction de la confiance qu'on souhaite avoir dans l'entité ainsi que de sa criticité au sein du logiciel, on combine ces différentes techniques de tests. Généralement on utilise le test fonctionnel basé sur les spécifications (formelles ou informelles) de l'entité pour détecter des erreurs de fonctionnement. Ces tests sont éventuellement complétés par des test structurels qui visent à s'assurer de la qualité du code qui la compose. On pourra ainsi effectuer des mesures de complexité du code ou assurer un critère de couverture structurel tel que la couverture de toutes les instructions.

Une méthode largement utilisée dans l'industrie consiste à effectuer une première passe de test fonctionnel et d'en retirer le taux de couverture du code obtenu pour un critère de couverture donné. Cette phase de test est alors complétée par une deuxième phase de test structurel de manière à obtenir un taux de couverture souhaité pour le critère de couverture le code de la fonction. La fonction est fournie avec son jeu de test et le pourcentage de couverture du code de l'entité pour le critère de couverture retenu pendant la phase de test précédente.

Le test d'intégration

La conception d'un logiciel peut être vu comme l'assemblage de briques unitaires simples (les entités) pour réaliser les fonctions complexes qu'on lui demande. Cet assemblage est effectué petit à petit suivant une stratégie pré-établie. A chaque étape de l'assemblage, on peut réaliser une phase de test, généralement pour s'assurer de la bonne communication des entités assemblés, c'est le test d'intégration.

Cette phase de test ne consiste pas à répéter exactement le même type de test que la phase unitaire sur des groupes d'entités. Durant cette phase on cherche à tester la manière dont chaque entité interagit dans son environnement d'utilisation.

Le test d'intégration n'est pas de nature statique. Il peut être effectué à chaque fois qu'une nouvelle entité est disponible. Il s'agit alors de tester sa bonne intégration parmi celles déjà testées. A cette fin, on utilise, comme pour la phase unitaire, des techniques de test fonctionnel ; cela dans le but de cibler le test sur les nouvelles fonctions du logiciel introduites par les entités dernièrement intégrées. De même les techniques fonctionnelles sont éventuellement complétées par des techniques structurelles lorsqu'il l'est jugé nécessaire. Généralement la couverture de toutes les instructions de l'ensemble des entités est considérée comme trop fastidieuse pour être réalisée. On lui préfère la couverture de tous les chemins du graphe d'appel de l'ensemble des entités.

Définition 2.15 *Le graphe d'appel d'appel d'un logiciel (ou d'un agrégat) est un graphe*

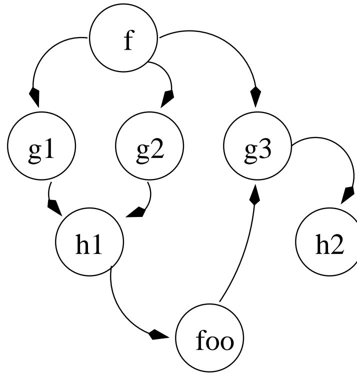

FIG. 2.3 – Une graphe d'appel

$Ga = \{N, A\}$ où tous les nœuds $n \in N$ représentent une fonction du logiciel et où tout arc $a \in A \mid n_1 \xrightarrow{a} n_2$ entre deux nœuds n_1 et n_2 de N, $a \in A \mid n_1 \xrightarrow{a} n_2$ indique qu'un appel de n_2 est présent dans le corps de n_1.

Sur le graphe d'appel où $N = f, g_1, g_2, g_3, h_1, h_2$ illustré sur la figure 2.3, les arcs représentent les appels entre entités. Par exemple, l'arc entre f et g_1 signifie que la fonction $f()$ appelle au moins une fois la fonction g_1.

La difficulté et les résultats du test d'intégration sont grandement conditionnés par la stratégie d'intégration choisie par les développeurs de logiciels. Il existe de nombreuses stratégies, chacune avec ses avantages et ses inconvénients. Le choix d'une stratégie est délicat, il dépend de l'architecture du logiciel mais également des composants disponibles pour chaque phase de test.

Il est important de se rappeler que le test d'intégration est effectué sur un ensemble incomplet d'entités. Cet ensemble n'a pas forcément de fonction précise au sein du logiciel. Il arrive qu'il fasse appel à des entités qui ne sont pas encore disponibles ou qu'il ne puisse être sollicité que par des entités qui ne sont pas encore intégrées.

Dans ce cas, un travail de préparation de l'ensemble des entités doit être réalisé avant le test. On construit de petits programmes de remplacement. Il s'agit de *lanceurs* lorsque il faut remplacer des entités qui invoquent celles à tester. On parle de *bouchons* lorsqu'on doit palier l'absence d'une entité appelée lors du test. Leur nature est semblable, il s'agit d'une représentation simplifiée de l'entité manquante.

Selon la stratégie d'intégration choisie pour le développement du logiciel, l'effort de préparation nécessaire à chaque phase de test d'intégration peut représenter une part importante de la totalité de l'effort de test. Les stratégies d'intégration les plus couramment utilisées sont :

- *La stratégie "big-bang"* où le test d'intégration ne commence que lorsque toutes les entités du logiciel sont disponibles. On les assemble alors toutes d'un coup et on teste le logiciel complet. Cette stratégie a l'avantage d'éliminer complètement le besoin de préparation au test. Malheureusement il s'en suit d'énormes difficultés pour identifier l'origine des erreurs mises au jour par le test.
- *La stratégie par incrément* où, après avoir défini l'ordre dans lequel on va intégrer les entités, on les ajoute une par un au logiciel à chaque étape du test. Ici la localisation des erreurs est facilitée : les erreurs détectées sont majoritairement situées dans la dernière entité intégrée. En contrepartie cette stratégie est fastidieuse, elle demande

un grand nombre d'étapes. De plus elle induit un travail de préparation important pour bouchonner les composants manquant à chaque étape de test.
- *La stratégie par agrégat* propose de regrouper les entités par groupe, un agrégat, pour fomer une fonctionnalité de haut niveau du logiciel. On crée de multiples sous-ensembles d'entités pour ensuite assembler ces sous-ensembles entre eux et créer le logiciel complet. Les erreurs rencontrées au cours du test sont plus difficiles à corriger que dans la stratégie par incrément. Une erreur détectée peut être localisée dans n'importe quelle entité de l'agrégat. Par contre elle limite le besoin de bouchons et de lanceurs en regroupant les entités s'appelant les unes les autres au sein du même agrégat.
  Elle diminue le nombre de cas de tests à produire pour solliciter le logiciel. Les entrées de tous les composants sont en rapport les unes avec les autres. Enfin elle est particulièrement adaptée aux architectures logicielles où les fonctionnalités sont indépendantes et facilement associées à un groupe d'entités restreint.

Quelque soit la stratégie d'intégration choisie, à l'exception de big-bang, il faut décider d'un ordre suivant lequel les entités seront intégrées pour former le logiciel. L'ordre a une importance majeure sur l'effort de préparation nécessaire au test ainsi que sur le type d'erreurs qui seront détectées plus facilement lors de la construction du logiciel. Les trois grandes manières d'intégrer les composant d'un logiciel sont les suivantes :

- Selon un ordre ascendant où les entités de plus bas niveau dans l'architecture de l'application sont intégrées en premier. Cette méthode limite par construction le nombre de bouchons à produire. Par contre les erreurs liées à l'architecture globale du logiciel ne pourront pas être détectées au plus tôt, elle n'apparaîtront que tard dans l'intégration des entités.
- Selon un ordre descendant où les entités de haut niveau sont prioritaires dans l'intégration. Elle représente mieux l'architecture complète de l'application et permet ainsi une meilleure réutilisation des jeux de test tout au long de la phase d'intégration. De plus, les jeux de test sont plus faciles à créer, ils sont liés à l'utilisation finale du logiciel et par conséquent ils sont plus naturels à produire pour un testeur. En contrepartie cet ordre nécessite un grand nombre de bouchons pour simuler le comportement "calculatoire" du logiciel. Ils sont également difficiles à écrire car ils doivent simuler le fonctionnement d'une grande partie du logiciel.
- Selon un ordre guidé par la criticité de l'entité où les entités jugées critiques dans le logiciel sont intégrées dès le début de la phase de test. Cet ordre a pour but de porter l'effort de test plus intensement sur les entités jugées importantes de manière à repérer au plus tôt les erreurs majeures susceptibles d'advenir. Cette méthode ne respecte pas l'architecture du logiciel et par conséquent demande un travail de préparation important pour simuler l'environnement des entités critiques. Une entité critique peut se situer n'importe où dans le logiciel et son exécution peut demander un grand nombre de données produites par les autres entités. Les lanceurs et les bouchons d'une entité critique vont être nombreux et difficiles à écrire pour un test efficace. Les jeux de test produits à chaque étape de l'intégration sont liés à une entité critique particulière qui faisait l'objet du test. Ils ne pourront pas être réutilisés dans les étapes ultérieures de l'intégration.

2.2 Méthodes de génération de données pour le test structurel

Le test structurel est utilisé en complément du test fonctionnel pour ajouter à la confiance dans le fonctionnement d'un logiciel, un niveau de confiance dans sa structure interne. Pour cela on fournit un niveau de couverture obtenu pour les entités du logiciel.

Etant donné un niveau de couverture à atteindre pour l'entité sous test, la génération de données peut être pratiquée de façon *aléatoire* ou *déterministe*. Dans le premier cas, l'évaluation de la couverture structurelle est souvent un calcul fait a posteriori : on évalue le niveau de couverture au fur et à mesure des exécutions de l'entité pour les données de test engendrées. Lorsque la couverture n'est pas jugée suffisante on produit de nouvelles données pour la compléter.

La génération déterministe de données de test consiste à choisir un certain nombre de composants de l'entité dont l'activation assure le niveau de couverture recherché. Ceci correspond à la génération *a priori* de données de test pour la couverture d'un critère structurel. Ce type de génération est un problème difficile.

Pour la génération a priori, les méthodes proposées peuvent être grossièrement classées en deux grandes catégories : *les méthodes statiques* et *les méthodes dynamiques*. Les premières reposent sur une analyse du code de l'entité tandis que les secondes tirent parti de multiples exécutions.

Voici deux exemples, parmi les plus connus, de techniques de génération déterministe de données des test :

- L'*exécution symbolique* [GWZ94, JBW$^+$94] est une méthode statique.
  Pour générer une donnée de test activant un composant donné d'une entité, on commence par sélectionner un chemin du GFC dont l'exécution va couvrir le composant.
  On calcule alors *le prédicat de ce chemin*, c'est à dire ses conditions d'exécution. Classiquement, cette construction se passe de la manière suivante :
  - On initialise la construction en considérant que toute variable x a une valeur courante initiale x_0.
  - toute affectation $x := expr$ est transformée en une égalité $x_{i+1} = \overline{expr}$ où i est l'indice de la valeur courante de x et $\overline{expr}$ est $expr$ où toute variable a été instanciée par sa valeur courante. La valeur courante de x devient x_{i+1}.
  - tout prédicat P (resp. négation de prédicat $\neg P$) d'une instruction de contrôle est remplacé par le prédicat $\overline{P}$ (resp. $\overline{\neg P}$) qui est le prédicat P (resp. $\neg P$) où toutes les variables ont été remplacées par leur valeur courante.

  Le prédicat du chemin est la conjonction de toutes ces formules.
  En résolvant chacun de ces prédicats, on détermine une donnée qui les satisfait : c'est la donnée de test. Cette donnée fera du chemin choisi un chemin d'exécution de l'entité.
  On peut noter qu'il n'est pas toujours possible de résoudre ces prédicats. Dans ce cas, le chemin n'est pas un chemin d'exécution. Il faut choisir un autre chemin assurant la couverture du composant.
  Dans le cas d'une entité sans boucle, l'ensemble des chemins est fini et peut être parcouru de manière exhaustive. Dans le cas d'une entité comportant des boucles, l'espace de recherche peut être infini. On n'explore alors qu'un sous-ensemble des

chemins généralement défini par un nombre fixé d'itérations dans les boucles. Par exemple les *chemins élémentaires du GFC* qui sont des chemins tels que chaque arc n'est emprunté qu'une seule fois.

– Korel a proposé une méthode de génération dynamique de données de test structurel [Kor96] qui s'appuie sur une instrumentation du code des entités et qui, par tatonnements successifs, génère une donnée assurant la couverture du composant c visé.

  Le principe consiste à d'abord générer aléatoirement un vecteur d'entrées pour l'entité et à l'exécuter. Si le composant c n'est pas exécuté par ce vecteur, le processus tente de modifier le chemin suivi pour atteindre c en se servant des dépendances de données [MS04] entre les variables.

  En cas d'échec, le processus identifie le composant problématique p (présent entre l'entrée de l'entité et c). p devient alors le sous but à atteindre.

  De cette manière, générer une donnée de test pour assurer la couverture d'un composant spécifique de l'entité se réduit à l'exécution d'un ensemble de sous buts (les composants problématiques intermédiaires) qu'il est nécessaire de réaliser pour couvrir le composant visé.

Par sa nature dynamique, cette méthode n'est pas sujette aux problèmes des chemins non exécutables. De plus elle ne nécessite aucun travail particulier pour traiter les instructions complexes comme les tableaux ou les pointeurs qui sont des points difficiles pour les méthodes statiques.

Toutefois, comme pour toute méthode dynamique, son utilisation peut être rendue difficile par l'entité qu'elle doit tester. Cette méthode nécessite un grand nombre d'exécutions des entités, ce qui peut poser problème lorsque cette exécution est longue ou qu'elle dépend d'un environnement d'exécution difficile à mettre en œuvre.

On remarque que toutes ces méthodes se basent sur une analyse directe ou indirecte du code des entités. Intuitivement, la complexité de cette analyse est liée aux nombres des chemins élémentaires de l'entité et aux nombres de conditions apparaîssant dans les prédicats de chemins.

Dans ce mémoire, nous nous intéressons à une méthode statique de génération automatique de test structurel a priori, orientée but, appelée Inka [Got00, GBR00]. Elle est basée sur la programmation logique avec contraintes [DCED96].

En s'appuyant sur la résolution de contraintes en programmation logique, elle permet, dans certaines circonstances, de donner une preuve de la non-atteignabilité d'un composant dans une entité.

La programmation logique avec contraintes ainsi que l'outil Inka sont présentés dans le chapitre 6. Cet outil sera la base des expérimentations de cette thèse (voir II).

Chapitre 3

Etude du problème

Dans ce chapitre, nous commençons par présenter le problème posé par les appels de fonctions dans le test structurel. Puis, l'analyse du problème nous oriente vers la proposition d'une méthode pour la prise en compte efficace de ces appels de fonctions [Gri02]. Le chapitre s'achève par une rapide étude bibliographique des techniques de transformations et d'approximations de code. La problématique qui sous-tend ces techniques est en effet proche de notre problème.

3.1 Appels de fonctions et bouchons

3.1.1 De la nécessité de traiter les appels de fonctions lors du test unitaire

Ce travail se place dans le cadre de la génération automatique de données de test structurel pour le test unitaire. Plus particulièrement, pour une entité donnée de type procédure ou fonction (nous utiliserons indifféremment l'un ou l'autre de ces termes), nous nous intéressons à la génération de données a priori, déterministe et statique, pour exécuter des composants choisis dans cette entité, en présence d'appels à d'autres entités.

Paradoxalement, comme le test unitaire recommande que chaque entité soit testée isolément, la prise en compte, dans l'entité testée, des appels aux autres entités n'est pas apparue comme un problème important.

La raison en est probablement que le code de ces dernières est considéré comme non disponible au moment du test unitaire. Le traitement des appels aux autres entités est donc reporté à l'étape d'intégration. D'où l'idée développée au paragraphe suivant de substituer des "bouchons simples" aux fonctions appelées.

Pourtant, si on cherche à produire des données de test pertinentes pour le test structurel d'une procédure f, il est souhaitable de pouvoir prendre en considération tous les appels d'entités rencontrés dans f. En effet, dans le cas contraire, on ne peut plus garantir que les niveaux de couverture obtenus pour les entités testées lors de la phase unitaire sont représentatifs de l'utilisation réelle de ces entités dans un agrégat ou plus généralement dans le logiciel. Ce type de test structurel ne peut pas être pratiqué lors du test d'intégration.

De plus, l'appliquation en test d'intégration des mêmes critères de couverture (arc, conditions, chemins, etc...) que ceux mis en œuvre pour le test unitaire peut rapidement

devenir irréaliste car le code de toutes les fonctions appelées doit être pris en compte et le nombre de composants à "couvrir" devient démesuré.

3.1.2 Inconvénients des bouchons simples

On s'intéresse à une fonction $f()$ qui en appelle une autre $g()$; on souhaite engendrer des données de test structurel a priori pour $f()$. L'appel à $g()$ est traité en introduisant un ou plusieurs "bouchons".

Définition 3.1 *Etant donné un composant spécifique à couvrir dans $f()$, le bouchon $g'()$ de $g()$ est une fonction qui se substitue à $g()$ et qui permet de déterminer une entrée de $f()$ pour un chemin de son GFC qui assure la couverture du composant sélectionné.*

★ **Exemple 3.1** : *Une fonction*

```
int f(int x){
   y = g(x);
   if(y == 5){/ * branch1 * /}
   else{/ * branch2 * /}
   return(1/(y − 7));
}
```

Nous appelons "bouchons simples", les bouchons construits dans le but unique de permettre la mise en œuvre du test structurel unitaire. Ces bouchons sont généralement construits à la main, et répondent à l'objectif d'exécuter un composant ciblé de l'entité appelante.

Considérons l'exemple 3.1. Pour exécuter *f()* et garantir que toutes ses branches soient couvertes, il est nécessaire de créer deux bouchons, $bg1$ et $bg2$, assurant chacun la couverture d'une branche de *f()* par exemple :

- $bg1$, quelle que soit la valeur du paramètre x, retourne la valeur 5 ;
- $bg2$, retourne une valeur tirée aléatoirement dans $]-\infty, 5[\cup]5, +\infty[$.

La construction de tels bouchons pour le test structurel unitaire est aisée mais leur utilisation pose deux problèmes majeurs :

- Tout d'abord ces bouchons ne sont pas nécessairement représentatifs des comportements de la fonction qu'ils remplacent. De ce fait, il peut arriver que des données de tests générées grâce à eux permettent de couvrir des composants de l'entité appelante qui ne l'auraient pas été avec la fonction originale. La couverture de l'entité sous test peut être surestimée.
  Imaginons, par exemple, que la fonction $g()$ de 3.1 ait été **int** $g($**int** $x)$ $\{return\ x\%5\ ;\}$. Dans ce cas de figure, la branche 1 de *f()* peut être couverte lorsque le processus de génération de données de test produit des données en utilisant le bouchon $bg1$ défini précédemment. Par contre, le processus de test sera dans l'impossibilité de générer une donnée couvrant le même composant en utilisant la véritable fonction $g()$.

– Par ailleurs, l'utilisation de bouchons retournant une valeur arbitraire peut révéler des erreurs inexistantes avec les fonctions originales.
  Toujours dans l'exemple 3.1, si la valeur tirée arbitrairement par le bouchon $bg2$ est 7, ce qui est correct vis à vis des attentes "structurelles" qu'on peut avoir pour ce bouchon, alors l'exécution de $f()$ avec la donnée de test produite pour couvrir sa branche 2 conduira à une erreur (division par 0) qui n'existe pas dans le logiciel.

Les techniques statiques de génération automatique de données des test (voir la section 2.2), sont soumises à ce type de difficultés. A notre connaissance, aucune de ces techniques ne propose de méthode pour prendre en considération les entités appelées lors de la génération de test.

On note que les techniques dynamiques de génération automatique de données de test n'utilisent pas de bouchons puisqu'elles exécutent le logiciel réel. Dans le cas de l'évaluation du niveau de couverture a posteriori (qui repose sur une génération non déterministe), le bouchon $g'()$ de $g()$ peut être plus simplement une fonction arbitraire qui se substitue à $g()$ et qui permet l'exécution de $f()$.

Nous allons proposer une méthode permettant de construire automatiquement de nouveaux bouchons qui répondent aux limitations des bouchons simples.

3.2 Analyse de la construction d'un bouchon et techniques associées

3.2.1 Analyse préliminaire du processus de création d'un bouchon

Notion d'environnement

Dans la suite, on utilise le terme "analyse" d'une entité, d'une fonction, d'une procédure, voir d'un bouchon, pour désigner le traitement effectué sur les instructions de cette entité, préparatoire à la génération de données de test a priori.

Ce traitement a pour but de déterminer un domaine pour les entrées de l'entité appelée, qui garantit qu'elle sera amenée à produire une sortie permettant la couverture d'un composant spécifique de l'entité appelante.

```
int f( int x) {
1   int y;
2   int z;
3   y = 2*x*x+3;
4   z = g(y);
5   if (z + y > 2)
6     y = y + z;
7   return y;
}
```

Plus généralement, pour construire un (ou des) bouchon(s) d'une procédure g partie intégrante d'un agrégat de fonctions et comme le processus de génération le ferait, on peut tirer des informations des différents appels à g.

Par exemple, de l'agrégat *"f qui appelle g"* de l'exemple ci-dessus, on peut tirer plusieurs informations au point d'appel à g dans f.

Une analyse rapide des dépendances de données entre les variables de f nous montre que la couverture de l'instruction *6* dépend du résultat de l'appel à g en *4*.

Avant d'entrer au cœur de l'analyse du corps de g, l'appel dans f nous donne des indications sur ce que son bouchon doit contenir pour permettre la mise en œuvre du test structurel.

Nous avons tout d'abord la séquence d'instructions :

```
int f( int x) {
1   int y;
2   int z;
3    y = 2*x*x+3;
4    z = g(y);
...
```

Le paramètre d'entrée de la procédure g est un entier y. Avant son utilisation dans g (en *4*), sa valeur est définie pour la dernière fois en *3*. Cette dernière définition de y nous apprend que le paramètre de g est > 3 à l'appel de g en *4*.

Ceci va nous permettre, pendant la construction d'un bouchon de g, de déterminer la valeur de certaines décisions dans des instructions de contrôle et ainsi d'obtenir un bouchon comportant moins d'instructions.

De même, pour atteindre l'instruction *6* de f, nous avons à examiner la séquence d'instructions :

```
...
3    y = 2*x*x+3;
4    z = g(y);
5   if (z + y > 2)
6      y = y + z;
...
```

La condition qui permet l'exécution de *6* est $z + y > 2$. Sachant que $z = g(y)$, l'appel à g conditionne la couverture du composant. On peut réécrire cette condition en $g(y) > -2 * x^2 - 1$ et la faire porter sur la sortie de g. Cette condition nous apprend que, pour exécuter le composant visé de la fonction appelante, le bouchon de g doit être capable de produire des sorties dans le domaine $]-2x^2 - 1, +\infty[$ mais également que toutes les sorties définies sur cet ensemble sont équivalentes dans le but de couvrir l'instruction *6* de $f()$.

Nous appelons ces informations, ici définies intuitivement, *un environnement* de g dans f.

Définition 3.2 *Informellement, nous appelons environnement d'une procédure à un de ces points d'appel, le regroupement des contraintes sur ses paramètres (pour que l'appel puisse avoir lieu) et les contraintes sur les valeurs de ses résultats (pour que le composant sélectionné dans la fonction appelante soit couvert).*

Il se décompose en un contexte d'appel et en un ou plusieurs objectifs de génération.

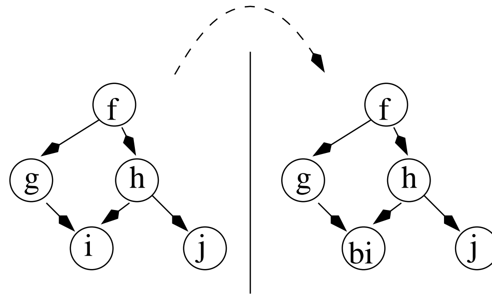

FIG. 3.1 – Graphe d'appel dans le cas d'un bouchon par entité appelée

Le contexte d'appel et les objectifs de génération seront définis formellement au chapitre 4.

Ces deux notions sont utiles pour simplifier et spécialiser le code des entités appelées, pour un appel particulier et de manière à obtenir un bouchon. Intuitivement, elles permettent de sélectionner, dans l'ensemble des comportements possibles de l'entité originale, ceux qui sont valides dans l'environnement et de regrouper ceux qui sont équivalents.

Différentes configurations de "bouchonnage" d'un agrégat

Nous utilisons les informations sur l'environnement des entités appelées pour éliminer des bouchons toutes les informations qui ne sont pas nécessaires à la couverture des composants de l'entité appelante. Le contexte d'appel et les objectifs de génération ouvrent la voie à différentes configurations pour la création de bouchons.

Selon qu'on tient compte ou non de l'environnement, nous distinguons trois schémas (ou configuration) de "bouchonnage" différents :

Un bouchon par entité

Dans ce premier schéma, un seul bouchon est créé pour l'entité appelée, indépendamment de ses divers environnements. Ainsi, tous les appels à l'entité seront remplacés par un appel vers cet unique bouchon qui se substitue complètement à l'entité originale dans l'agrégat.

L'effet de ce schéma sur le graphe d'appel est illustré par la figure 3.1. On modifie l'agrégat original pour représenter l'utilisation d'un bouchon unique $bi()$ pour l'entité appelée $i()$. Les appels effectués par les entités appelante $g()$ et $h()$ sont redirigés vers $bi()$.

L'avantage de ce schéma est qu'on ne crée que peu de bouchons pour l'agrégat. Le nombre de bouchons correspond au nombre d'entités susceptibles d'être appelées pendant le processus de test. Cette configuration est intrinsèquement statique : la création de bouchons peut être complétement dissociée du processus de génération de données de test.

Un bouchon par couple entité appelée - entité appelante

Ce schéma permet de créer un bouchon par entité appelée dans chaque entité appelante. Ainsi, pour une même entité appelée, on dispose d'autant de bouchons qu'il y a

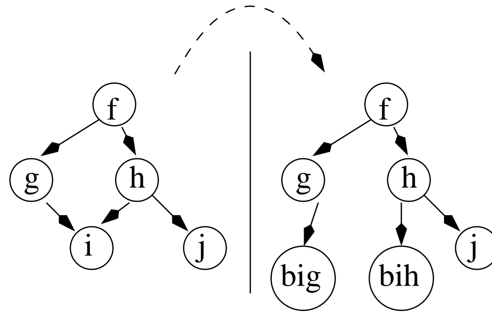

FIG. 3.2 – Graphe d'appel dans le cas d'un bouchon par couple entité appelante - entité appelée

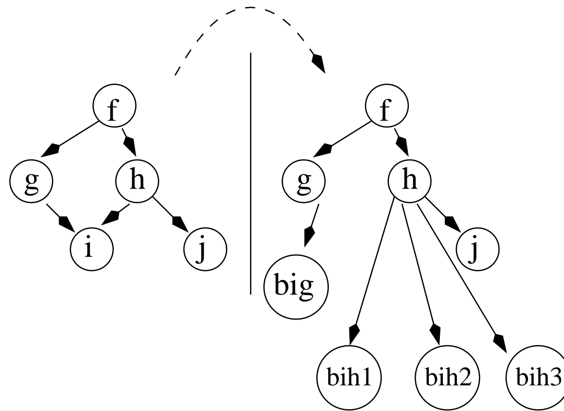

FIG. 3.3 – Graphe d'appel dans le cas d'un bouchon par appel

d'entités l'appelant.

Ce schéma est illustré par la figure 3.2. L'entité $i()$ est appelée par les entités $g()$ et $h()$. Par conséquent, l'agrégat original est modifié pour introduire les deux bouchons $big()$ et $bih()$ issus de $i()$. Les appels effectués par $g()$ et $h()$ sont redirigés vers leurs bouchons respectifs.

Ce schéma présente l'avantage de tenir compte des spécificités des entités appelantes. Cependant, la création de bouchons est nettement plus compliquée que celle du schéma précédent ; il faut notamment construire l'environnement de chaque entité appelée dans l'entité appelante. Il y a un environnement par appel.

Ces différents environnements doivent ensuite être regroupés pour n'en former qu'un par entité appelée. Cet environnement global est alors utilisé pour construire le bouchon.

La synthèse de tous les environnements d'appel pour une entité ne peut être réalisé autrement que statiquement, cette configuration est donc aussi statique.

Un bouchon par appel

Le dernier schéma va plus loin dans l'affinement des bouchons. Ici on produit un bouchon par environnement.

Chaque entité dispose d'un nombre de bouchons au maximum égal au nombre de fois où elle est appelée dans l'agrégat. Ce schéma est illustré sur la figure 3.3. Dans cet exemple, nous avons considéré qu'il y a un appel à l'entité $i()$ dans $g()$ et trois de

ces appels dans $h()$. Cette configuration se retrouve dans la modification effectuée sur l'agrégat. Il y a un unique bouchon $big()$ de $i()$ pour $g()$ et trois bouchons spécifiques $bih1()$, $bih2()$ *et* $bih3()$ de $i()$ pour les appels issus de l'entité $h()$.

L'avantage de ce schéma est une prise en compte optimale de l'utilisation de l'entité ; comme on calcule un environnement par appel, chaque bouchon est représentatif des comportements originaux de la fonction pour cet appel.

Bien que précise, cette solution est coûteuse : elle nécessite de construire un environnement pour chaque appel de procédure et un bouchon par environnement.

Pour obtenir cette configuration de manière efficace, on peut s'appuyer sur une création dynamique des bouchons au moment de l'appel. Ainsi, on dispose des domaines des entrées des entités appelées sans avoir à les calculer explicitement. Malheureusement, la création dynamique des bouchons pendant la génération de test a l'inconvénient de nécessiter une modification du processus de génération de données.

A contrario, une création statique nécessite un calcul explicite des environnements mais permet l'indépendance par rapport au processus de génération.

Construction d'un bouchon à partir du corps d'une fonction

L'utilisation du corps des fonctions appelées simplifiées par leurs environnements en tant que bouchon ne permet pas la mise en œuvre du test structurel unitaire. Les bouchons ainsi obtenus ne sont pas suffisamment simples à analyser.

D'autant plus qu'en pratique, un environnement n'est calculable précisement que lorsqu'aucun appel d'entité n'intervient dans la définition des variables qui le compose.

C'est pourquoi, il est nécessaire, en plus du calcul des environnements, de simplifier le code des entités appelées pour former un bouchon.

Un bouchon suffisant pour le test structurel d'une entité appelante doit principalement répondre à deux critères :

1. tout d'abord, pour le processus de génération de données, l'analyse du bouchon doit être plus simple que celle de l'entité qu'il remplace. Le sens qu'on peut donner à cette simplicité dépend du processus de test effectivement utilisé. Intuitivement, il peut s'agir d'un bouchon formé d'un plus petit nombre d'instructions ou encore dont le nombre de chemins d'exécution est moins important. Nous définirons plus précisément cette notion de simplicité dans le chapitre 4.1.
2. Pour un ensemble d'entrées fournies, les sorties calculées par le bouchon doivent être équivalentes du point de vue du critère structurel à celles qu'aurait produites l'entité originale. Cette équivalence est garantie par l'utilisation des objectifs de génération du bouchon dans son environnement.

Ces deux critères semblent contradictoires, le premier nécessitant une simplification du code de l'entité tandis que le second impose l'équivalence des sorties calculées. La simplification du code de l'entité ne peut donc pas être destructrice. Le code produit par la simplification doit être équivalent au code original pour le calcul des données.

Dans le chapitre 4, nous proposons une méthode de génération automatique de bouchons pour un agrégat de fonctions. Ces bouchons répondent au critère de suffisance pour le test structurel.

Cette méthode utilise à la fois le flot de données pour calculer l'environnement des bouchons et le flot de contrôle des entités appelées pour simplifier leur code.

Notre approche est également basée sur des techniques heuristiques de simplification du code des fonctions. Ces heuristiques sont inspirées de techniques connues de simplification et de transformation de code.

3.2.2 Techniques de transformations et d'approximations de code

Il existe, dans la littérature, des techniques spécifiques de transformation et de simplification de code. La section 3.2.2 présente les différentes techniques dont nous nous sommes inspirés. En particulier, les techniques d'évaluation partielle [JGS93] et d'interprétation abstraite [CC77b, P.78, CP75, CC77a, CC76] nous ont guidés dans le calcul de l'environnement d'un bouchon tandis qu'une adaptation à notre contexte de travail de la technique de slicing [Wei84, Luc01, Tip95, RT96, HRB88, SHR99] nous a mené à notre méthode de transformation du code des entités appelées.

Techniques de transformations : Slicing et Evaluation partielle

La transformation de programmes permet d'obtenir un nouveau code en opérant des modifications sur le code d'un programme existant. On s'intéresse particulièrement à deux types d'opérations effectuées sur le code : *la décomposition* et *la spécialisation*.

La décomposition effectue une coupe d'un programme afin d'en conserver les parties répondant à un critère préétabli. Il s'agit d'une modification purement structurelle. La technique la plus courante s'appelle *Slicing* [Wei84].

L'évaluation partielle est une technique de spécialisation. Elle opère une transformation de nature comportementale sur un programme générique et le spécialise pour l'adapter à une utilisation particulière.

Slicing

Le slicing (tranchage) est une technique, originalement proposé par Mark Weiser [Wei84], qui permet de décomposer automatiquement un programme en analysant son flot de contrôle et son flot de données pour isoler un comportement spécifié par un critère de découpe. Le résultat d'une telle décomposition, pour une procédure et un critère, est un sous-programme qui possède le comportement original recherché.

Toutes les instructions et les branches de la procédure originale qui n'ont pas d'intérêt vis à vis du critère n'apparaissent pas dans le sous-programme.

En pratique, cette technique permet d'isoler, dans le corps de la procédure, les parties du code influant sur la valeur d'une variable ou d'un groupe de variables à un point donné de la procédure ; c'est le critère de décomposition, $C = (P, V)$, où P est un point du programme, par exemple une instruction, et V un ensemble de variables d'intérêt au point P.

Le calcul d'une tranche consiste à construire récursivement l'ensemble des noeuds et des arcs du graphe de flot de contrôle ayant un impact sur la valeur des variables V de C au point P en remontant le graphe de flot de contrôle de la fonction à partir de ce point.

Cette méthode de slicing est appelée intra-procédurale, elle permet de créer la tranche d'une entité indépendament des autres entités appelées ou appelantes avec lesquelles elle

pourrait être en interaction.

La méthode proposée par Weiser pour étendre la tranche d'une procédure P seule à une tranche inter-procédurale de P appelée et appelante consiste à produire une tranche comme l'union des tranches intra-procédurales des fonctions appelantes et appelées par P.

L'inconvénient de cette méthode inter-procédurale est qu'elle ne produit pas des tranches précises. En effet, l'union des tranches peine à prendre en considération le contexte d'appel des fonctions.

S. Horwitz, T. Reps et D. Binkley ont proposé une autre méthode de Slicing inter-procédurale [HRB88, SHR99]. Elle se base sur l'utilisation d'un graphe particulier : le System Dependence Graph [HRB88]. Ce graphe est une extension du Program Dependence Graph dont les travaux de Ottenstein et Ottenstein [OO84] ont montré l'adéquation avec la technique de Slicing.

Le System Dependence Graph permet de représenter des programmes dans lesquels on trouve des procédures et des appels de procédures quand le Program Dependence Graph ne permet qu'une représentation monolithique des programmes. Il est constitué d'un Program Dependence Graph pour la fonction principale du programme et de Procedure Dependence Graphs pour toutes les autres procédures du programme. Le lien entre ces différents graphes est réalisé grace à des arcs spécifiques au System Dependence Graph, ils peuvent être de deux sortes :

1. Les arcs qui représentent une dépendance directe entre le lieu de l'appel et la procédure appelée. Ces arcs relient, via de nouveaux noeuds, les différents graphes de dépendance lors de l'appel.
2. Les arcs qui représentent une dépendance transitive due aux appels de procédures. Ces arcs sont calculés à partir d'une grammaire attribuée, appelée grammaire de liens. Cette grammaire représente la structure d'appel du programme, elle comporte un non terminal pour chaque procédure.

   Les attributs de la grammaire correspondent aux paramètres des procédures : les paramètres d'entrées sont hérités et les sorties sont synthétisées.

De plus de nouveaux noeuds sont introduits avant et après chaque graphe de dépendance afin de représenter :
- L'entrée dans la procédure appelée.
- Le passage de paramètres. Un noeud temporaire est utilisé pour chaque paramètre de façon à mettre en relation les paramètres effectifs de l'appel et paramètres formels de la procédure appelée.
- La même mise en relation est effectuée pour le renvoi des résultats à la procédure appelante lorsque c'est nécessaire.

Ces nœuds sont reliés entre eux par des arcs spécifiques *parameter-in, parameter-out et call*.

Les transitions du graphe associé au code de la fonction représentent les différentes dépendances entre variables. Pour isoler la partie du code relative au critère de découpe et calculer la tranche inter-procédurale, on parcourt ces transitions dans un ordre précis, tout en marquant les noeuds du graphe suivant l'algorithme de découpe. Le slicing inter-procédural d'un procédure P pour un sommet s est réalisé en deux phases :

1. Dans la première phase on parcourt le System Dependence Graph et on identifie tous les noeuds qui peuvent atteindre s depuis P ou depuis une procédure qui appelle P. On ne suit pas les arcs de type *parameter-out*, par conséquent l'analyse ne "descend" pas dans les procédures appelées par P. Pourtant, grâce aux arcs de dépendances transitives, on peut détecter un noeud de P qui ne peut atteindre s qu'à travers l'appel d'une procédure sans avoir besoin de l'analyser.
2. La deuxième phase permet l'identification des noeuds qui permettent d'atteindre s depuis des procédures appelées par P ou depuis des procédures qui appelent des procédures appelent transitivement P. Ici on ne remontera pas les arcs de types *parameter-in et call* de cette manière on ne "remonte" pas dans les procédures appelantes ; les arcs de dépendances transitives entre les paramètres effectifs d'entrées et de sorties rendant l'analyse de la fonction appelante superflue.

Evaluation partielle

L'évaluation partielle [JGS93] est une technique de transformation et de spécialisation de programmes. Elle fournit un paradigme unificateur pour un large spectre d'activités dans des domaines divers tels que l'optimisation des programmes, l'interprétation, la compilation et autres formes de génération de programmes [CD91].

L'évaluation partielle permet d'extraire d'une fonction p à plusieurs paramètres une fonction $p1$ possédant moins de paramètres que la précédente. Pour cela certains paramètres sont fixés à une valeur choisie par l'utilisateur. Ces paramètres statiques sont ensuite propagés dans le corps de la fonction p originale pour obtenir la fonction $p1$ spécialisée pour l'utilisation induite par les valeurs choisies. Ce fonctionnement est représenté sur la figure 3.4.

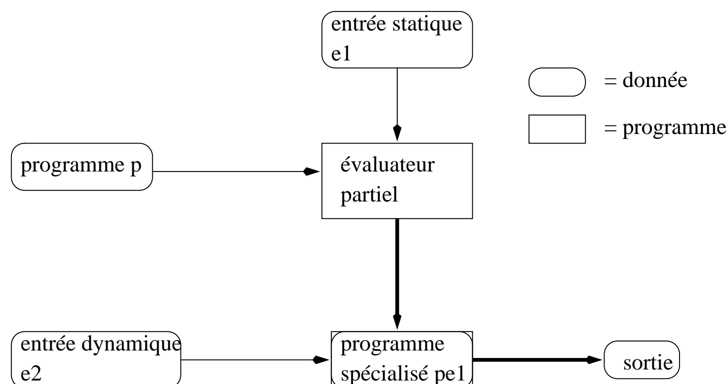

FIG. 3.4 – Un évaluateur partiel

L'évaluation partielle vise à produire des programmes efficaces à partir de programmes généraux de façon totalement automatique. Intuitivement, elle est effectuée en exécutant tous les calculs de p qui dépendent uniquement de $e1$ et en générant du code pour tous les calculs qui dépendent de la donnée dynamique $e2$. L'évaluation partielle est une technique mélant l'exécution et la génération de code. Elle repose sur trois sous techniques propres à la transformation de programmes :

1. L'exécution symbolique.
2. Le dépliage des appels de fonctions.

3. La spécialisation de point de programmes.

De manière générale, cette technique est utilisée pour améliorer les performances en temps d'exécution. De la même façon que le temps nécessaire à l'interprétation d'un programme est souvent plus long que celui nécessité par la compilation et l'exécution d'un programme équivalent, l'évaluation partielle d'un programme et l'exécution de la version spécialisée sur les données dynamiques restantes est souvent plus rapide que l'exécution du programme général. Sur l'exemple de la figure 3.4 c'est particulièrement vrai si la donnée dynamique $e2$ change beaucoup plus souvent que la donnée rendue statique $e1$: $t_{peval(p,e1)} + t_{p_{e1}(e2)} < t_{p(e1,e2)}$.

L'évaluation partielle peut aussi être utilisée lorsqu'on cherche à résoudre tous les problèmes similaires d'une même classe. Ecrire un petit programme efficace pour chaque problème poserait des problèmes de maintenance. Un programme général paramétrable permet d'assurer une maintenance aisée au détriment de l'efficacité de la solution. On peut alors utiliser l'évaluation partielle pour générer automatiquement de petits programmes efficaces en fixant les paramètres d'un programme général facile à maintenir.

Technique d'approximation : l'Interprétation Abstraite

Le concept qui soutend les techniques d'approximation est le principe d'abstraction. On le rencontre dans toutes les disciplines où il y a une notion de calcul. Son objectif est de pallier au problème de la complexité au prix d'un perte d'information. La technique la plus connue dans ce domaine est l'*interprétation abstraite* [CC76, CC77b]. Elle permet l'analyse de programmes complexes grâce à l'approximation de leurs comportements.

L'interprétation abstraite est une théorie de l'approximation de sémantiques de langages (de programmation ou de spécification). Elle a la faculté d'analyser de manière statique les propriétés dynamiques des programmes. Cette technique peut être considérée comme une extension des techniques de compilation qui permettent aux programmeurs de prédire le comportement futur de leurs applications, avant même leur exécution.

L'interprétation abstraite permet de formaliser l'idée qu'une sémantique est plus ou moins précise selon le niveau d'abstraction auquel on se place. Elle permet, également, de dériver les propriétés dynamiques des données à partir du code source du programme, et les appliquer pour l'analyse des propriétés dynamiques spécifiques. Elle est basée sur le principe d'évaluation abstraite, qui formalise une perte d'information en permettant d'obtenir une sémantique moins précise que la sémantique du programme d'origine.

L'interprétation abstraite est définie comme étant une interprétation symbolique d'un programme en associant aux variables des valeurs abstraites au lieu des valeurs concrètes utilisées lors de son exécution. Ainsi, chacune des opérations élémentaires du programme est interprétée (évaluée) selon ces valeurs.

Un exemple d'évaluation abstraite est illustré par la règle des signes qui permet de prédire le signe du résultat d'une opération arithmétique avant même le calcul de la formule. Si, par exemple, x est un nombre positif et y est un nombre négatif. D'après les tableaux 3.2.2 et 3.2.2, on peut déduire que la formule $x * x + y * y$ est positive avant même la calculer.

Définitions

Une *valeur abstraite* dénote une abstraction d'un ensemble de valeurs concrètes (définie

TAB. 3.1 – Evaluation abstraite des signes pour l'opération d'addition

- + -	+	-
+	+	?
-	?	-

TAB. 3.2 – Evaluation abstraite des signes pour l'opération de multiplication

- * -	+	-
+	+	-
-	-	+

en extension) ou des propriétés de cet ensemble (définie en intention) qui satisfont un certain nombre de propriétés dynamiques [CC76].

Une *fonction d'abstraction*, notée α est une fonction qui fait correspondre à un ensemble de valeurs concrètes leur valeur abstraite associée.

La *valeur abstraite nulle* est la valeur abstraite associée à un ensemble vide. $\overline{\phi} = \alpha(\phi)$, où ϕ est un ensemble vide de valeurs concrètes.

Par exemple, dans le cas où les valeurs concrètes seraient des valeurs entières appartenant à l'ensemble $S \subseteq Z$, la valeur abstraite que la fonction d'abstraction α leur associe est l'intervalle défini par :

- $\alpha(S) = [Min(x), Max(y)]$ où $x \in S$ et $y \in S$

Par exemple :

- $\alpha(\{-1, 2, 20\}) = [-1, 20]$
- $\alpha(\{1, 2, 3, 4, ...\}) = [1, +\infty]$

La *fonction inverse d'une abstraction*, notée γ est la fonction qui fait correspondre à une valeur abstraite un ensemble de valeur concrètes.

- $\gamma([a, b]) = \{x \mid (x \in Z) \cup (a \leq x \leq b)\}$

Si on appelle V_C et V_A respectivement l'ensemble des valeurs concrètes et l'ensemble des valeurs abstraites. Les deux fonctions α et γ sont définies de telle sorte que :

1. $\forall s \in V_C, s \subseteq \gamma(\alpha(s))$
2. $\forall v \in V_A, s = \alpha(\gamma(v))$

La première formule signifie que l'évaluation abstraite s'accompagne d'une perte d'informations.

Propriétés

- Par correspondance à l'ensemble des valeurs concrètes, l'union des valeurs abstraites est définie pour chaque évaluation abstraite.
- L'union des valeurs abstraites entières, notée $\overline{\cup}$ est définie comme suit : $[a_1, b_1]\overline{\cup}[a_2, b_2] = [Min(a_1, a_2), Max(b_1, b_2)]$
- L'union des valeurs abstraites est associative, commutative et idempotente.
- Par correspondance à l'ensemble des valeurs concrètes, l'inclusion des valeurs abstraites est définie pour chaque évaluation abstraite.
- L'inclusion des valeurs abstraites entières, notée $\overline{\sqsubseteq}$, est donnée par la formule suivante : $v_1 \overline{\sqsubseteq} v_2 \Leftrightarrow v_1 \cup v_2 = v_2$

– Deux valeurs abstraites sont comparables si les ensembles de valeurs concrètes correspondants sont comparables.
– L'ensemble des valeurs abstraites muni de l'opération d'union définit une structure de groupe.

Un *contexte abstrait* est l'ensemble de paires (i, v) où i dénote un identificateur de variable et v dénote sa valeur abstraite à un point quelconque du programme. Une propriété importante d'un contexte abstrait (i, v) est que toute exécution effective du programme donne des valeurs pour i qui appartiennent à $\gamma(v)$.

Le *contexte abstrait nul* par rapport à une variable i, noté $\Phi(i)$, est le contexte abstrait associé à une valeur abstraite nulle. On note : $\Phi(i) = \overline{\phi}$.

L'union (resp. l'élargissement) de deux contextes abstraits C1 et C2, notée $\overline{\overline{\cup}}$ (resp. $\overline{\overline{\bigtriangledown}}$) est utilisée pour exprimer le contexte résultant d'instructions conditionnelles (resp. d'une boucle). Ils sont définis par les deux formules suivantes [CC76] :
– $C_1 \overline{\overline{\cup}} C_2 = \{(i, v) \| (i \in I) \wedge (v \in V_A - \{\})) \wedge (v = C_1(i) \overline{\cup} C_2(i)\}$
– $C_1 \overline{\overline{\bigtriangledown}} C_2 = \{(i, v) \| (i \in I) \wedge (v \in V_A - \{\})) \wedge (v = C_1(i) \overline{\bigtriangledown} C_2(i)\}$

Interprétation Abstraite et approximation

L'approximation d'un programme par interprétation abstraite [CP75] s'effectue en deux étapes : tout d'abord, on construit son graphe de flot de contrôle ; ensuite on applique l'algorithme de l'interprétation abstraite sur le graphe.

Pour chaque programme impératif, un graphe de flot de contrôle peut être construit à base des composantes élémentaires graphiques suivantes :
– un nœud d'entrée,
– un ou plusieurs nœud(s) de sortie,
– des nœuds d'affectation,
– des nœuds de test,
– des nœuds de jonction simples,
– des nœuds de jonction de boucles.

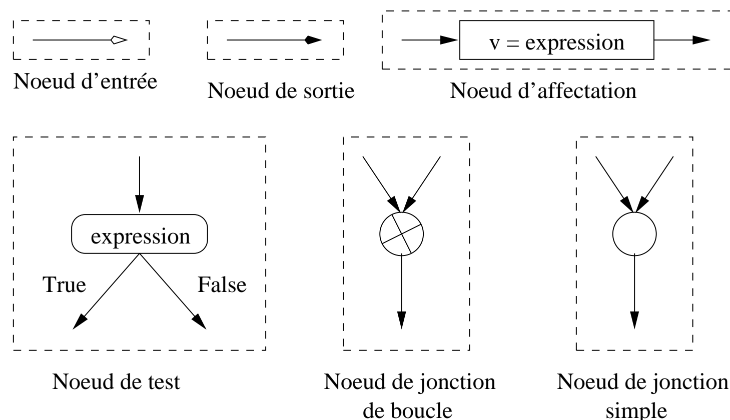

FIG. 3.5 – Nœuds d'un graphe de contrôle abstrait

L'interprétation abstraite d'un programme impératif [CP75] est conduite par une série d'évaluations (interprétations) abstraites atomiques. Chaque évaluation atomique correspond à l'interprétation abstraite d'une unité élémentaire (nœud). A cet effet, une fonction

$\Im$ est définie, qui pour chaque unité élémentaire (nœud) et un contexte d'entrée C, produit un contexte de sortie $\Im(n, C)$ ou deux dans le cas d'un nœud de test.

L'application $\Im$ doit être une abstraction correcte de l'exécution des unités élémentaires. Elle est définie, pour chaque type de nœuds.

L'interpréteur abstrait

L'interprétation abstraite d'un programme consiste à soumettre le graphe de flot de contrôle à un interpréteur abstrait. La tâche d'un interpréteur abstrait est de fournir pour chaque point (arc) du graphe son contexte abstrait. Il commence toujours par un contexte vide sur tous les arcs. Pour chacun des nœuds, l'interpréteur effectue la transformation adéquate qui spécifie les contextes de sortie en fonction des contextes d'entrée.

L'algorithme d'interprétation [CP75] effectue ces transformations jusqu'à ce que tous les contextes soient stables. Un contexte est dit stable si la transformation opérée au niveau du nœud en question n'entraîne aucun changement au niveau des contextes de sortie.

Chapitre 4

Un bouchon comme une hiérarchie d'approximations

Ce chapitre s'articule autour de trois grandes sections. De manière générale, nous présentons le plus souvent possible les nouvelles notions de manière intuitive avant de les définir formellement.

Nous commençons par une synthèse des besoins retenus après le chapitre 3 et une proposition pour répondre à ces besoins.

Les deux sections suivantes détaillent les heuristiques nécessaires à la réalisation de cette proposition. Il s'agit de créer un modèle pour chaque entité appelée de l'agrégat puis de spécialiser ce modèle à l'aide de l'environnement des entités pour obtenir les bouchons spécifiques au test structurel unitaire.

4.1 Approche retenue

4.1.1 Synthèse des besoins

L'unité de travail que nous considérons est un agrégat d'entités s'appelant les unes les autres ; il s'agit d'un ensemble d'entités à tester unitairement.

Réaliser le test unitaire de ces entités consiste à extraire, de l'agrégat, le graphe d'appel dont l'entrée est l'entité à considérer. Par exemple, pour l'agrégat de la figure 4.1, réaliser le test unitaire de la fonction $g2()$ consiste à extraire le graphe d'appel de la figure 4.2 et à construire les bouchons de toutes fonctions appelées de ce graphe.

Par rapport au cadre idéal où toutes les entités du logiciel pourraient être analysées simultanément, cet ensemble de travail omet volontairement toutes les entités appelant ou étant appelées par le système sous test, c'est à dire toutes les interactions avec les entités extérieures au système considéré.

Nous souhaitons être en mesure de produire des données de test réalistes pour chaque entité de l'agrégat. Le but est d'assurer que les niveaux de couverture de chaque entité obtenus par exécution de ces données soient conformes aux comportements possibles de ces entités dans cet agrégat.

Le problème qui se pose alors est d'éviter de subir la complexité d'une analyse exhaustive de toutes les entités appelées.

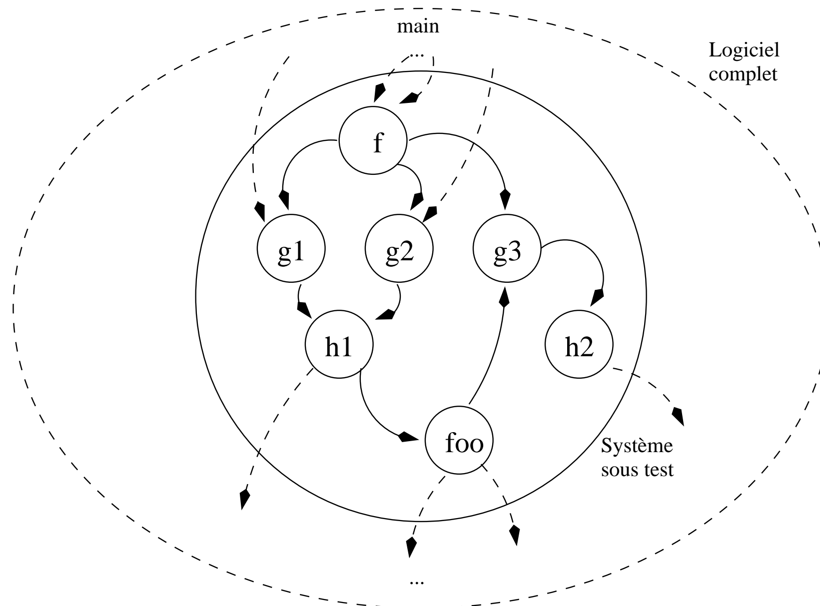

FIG. 4.1 – Le système sous test complet

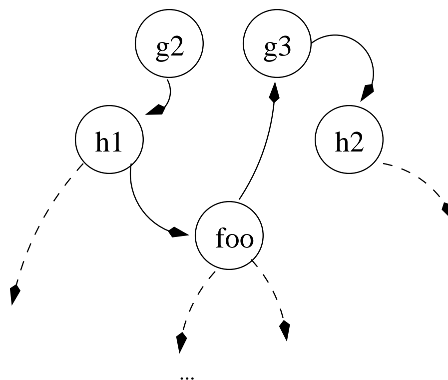

FIG. 4.2 – Le graphe d'appel pour le test unitaire de $g2()$

Dans ce but, nous nous proposons de produire automatiquement des bouchons pour le test unitaire des entités de l'agrégat, construits de sorte à être suffisants pour les besoins du test structurel. De ce fait, ils ne sont pas nécessairement une représentation exacte des entités qu'ils remplacent. Toutefois, leurs sorties doivent pouvoir être produites par les entités qu'ils remplacent si elles étaient utilisées dans un environnement similaire.

Ces bouchons constituent une situation médiane entre les bouchons simples (voir chapitre 3.1.2) qui ne représentent pas les entités, et les entités dans leur globalité qui sont trop complexes pour être analysées.

Quantitativement, la construction automatique de bouchons pour le test structurel unitaire à partir des informations issues de l'agrégat sous test va permettre de diminuer le coût global du test. Non seulement les bouchons ne sont plus construits à la main par le testeur mais ils sont également conçus pour corriger les principaux défauts des bouchons simples.

Pour la création des bouchons, toutes les entités de l'agrégat sont connues. Il s'agit d'extraire de l'agrégat, un modèle des entités appelées *suffisant* pour la couverture struc-

turelle des entités appelantes.

Dans ce but, nous avons à disposition le code des entités appelées mais également leur environnement d'appel précis par l'intermédiaire du code des entités appelantes.

L'analyse des bouchons issus des modèles des entités appelées va conduire le processus de génération à produire des données qu'il aurait pu générer en analysant le corps des entités appelées originales.

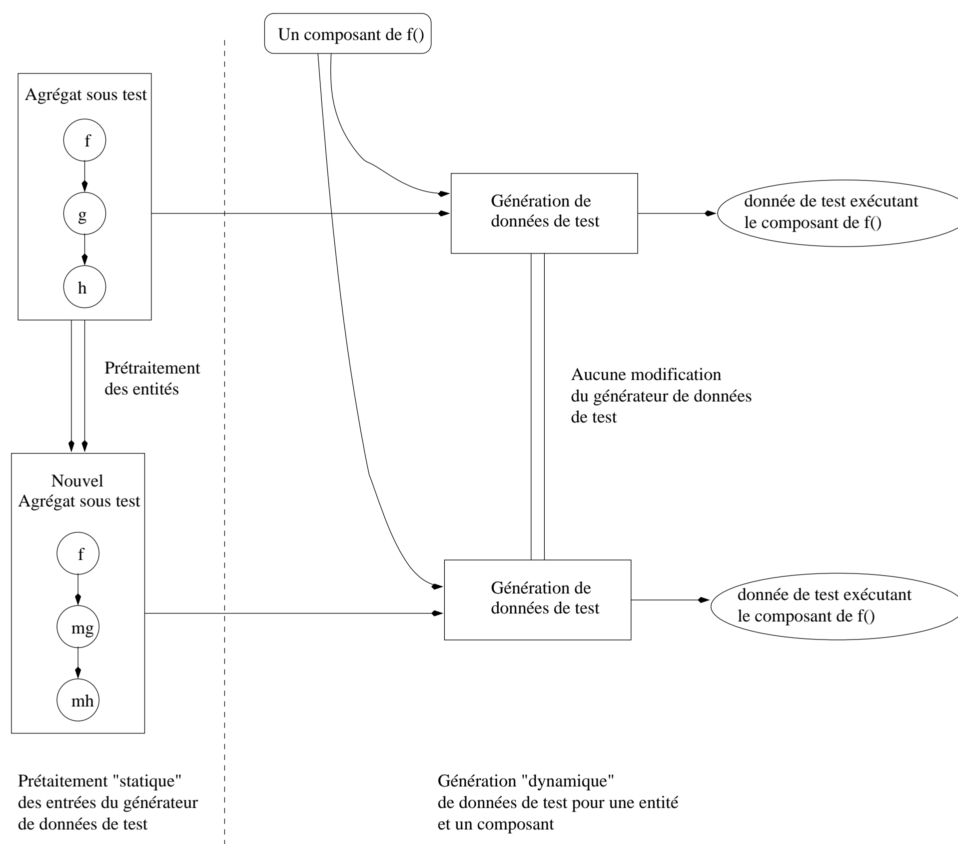

FIG. 4.3 – Processus de génération automatique de cas de test structurel

Enfin, pour rendre la génération automatique de bouchons pour un agrégat indépendante du processus de génération de données de test auquel il sera soumis, une configuration de création statique est la meilleure option. Avec une telle configuration, la création automatique des bouchons est effectuée sur l'agrégat d'entités, elle consiste à modifier son graphe d'appel. Cette modification est indépendante de processus de génération.

Cet état de fait est illustré par la figure 4.3. La partie supérieure de cette figure montre le fonctionnement schématique de la génération d'une donnée de test pour la couverture d'un composant d'une fonction f appartenant à l'agrégat sous test. Cet agrégat est composé de fonctions f, g, h ; il est considéré comme l'entrée du processus de génération. Celle-ci, couplée avec un composant à couvrir de f, va amener le processus de génération à produire une donnée de test qui assure la couverture du composant.

La partie inférieure de cette figure montre le même fonctionnement sur le nouveau graphe d'appel de l'agrégat de fonctions modifié par la création automatique de bouchons pour le test structurel unitaire. Le prétraitement statique de l'agrégat d'entités n'entraîne aucune modification du processus de génération de données de test.

Dans la suite de ce document, on parlera :
– des *entrées* d'une entité pour désigner les paramètres utilisés. Ces paramètres regroupent aussi bien les variables locales à l'entité que les éventuelles variables globales utilisées par l'entité ;
– des *sorties* d'une entité pour désigner l'ensemble des paramètres (locaux et globaux) dont les valeurs sont définies par une exécution de l'entité.

4.1.2 Complexité de la génération d'une donnée de test

Intuitivement, parmi les données de test produites à la main par un testeur dans le but de couvrir un composant donné, certaines sont plus faciles à calculer que d'autres. Ce critère de facilité dépend essentiellement de la nature et du nombre de points de choix (nœuds de décisions du graphe de flot de contrôle) dont il faut déterminer la valeur pour exécuter ce composant.

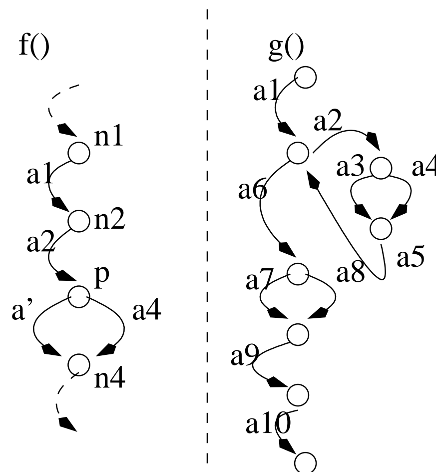

FIG. 4.4 – Graphes de flot de contrôle de f et g

Soit f et g deux fonctions (avec f appelant g) et G_f, G_g leurs graphes de flot de contrôle respectifs. Soit PG_f (resp. PG_g) l'ensemble de tous les chemins d'exécution à travers G_f (resp. G_g). Soit p un noeud de décision de f faisant intervenir un appel à g, et d l'ensemble des données de test permettant la couverture d'un arc a' placé après p dans l'exécution de f. Intuitivement, les chemins d'exécution de PG_g qui sont activés quand on fournit à f les données de d ont une complexité différente.

Cette situation est illustrée sur la figure 4.4. Intuitivement, si les chemins d'exécution $a_1, a_6, a_7, a_9, a_{10}$ et $a_1, a_2, a_3, a_5, a_2, a_4, a_5, a_6, a_8, a_9, a_{10}$ de g permettent d'exécuter a' de f, il sera plus facile pour le processus de test d'analyser le premier plutôt que le second.

La complexité des chemins de l'entité appelée n'a pas d'impact direct sur les données qu'ils sont amenés à produire. Pour construire un bouchon suffisant pour le test structurel, nous allons donc mettre en avant les chemins les plus simples à analyser. Les bouchons construits de cette manière vont forcer le processus de génération de test à analyser d'abord les chemins d'exécution simples avant de considérer les plus compliqués.

On peut résumer cet état de fait par ces deux constatations :

– Du point de vue de l'analyse des instructions par le processus de test, les chemins qui composent une entité ne sont pas équivalents. Certains chemins sont plus difficiles à analyser que d'autres.
  Intuitivement, si on considère que l'analyse de toutes les instructions élémentaires d'un langage représente la même difficulté, alors plus un chemin est long, plus il est complexe à analyser. La complexité d'un chemin dans un graphe de flot de contrôle est directement liée au nombre d'instructions qui le compose.
– Du point de vue de la couverture structurelle d'une entité appelante, seule compte la sortie produite par cette entité et les composants de l'entité appelante qu'elle permet d'exécuter (les chemins empruntés dans l'entité appelée sont indifférents).

Autrement dit, pour atteindre un composant donné de l'entité appelante, le processus de test peut choisir indifféremment un chemin de l'entité appelée qui produit une sortie suffisante.

Dès lors, il est possible de réorganiser l'entité appelée de façon à mettre en avant les chemins (ou ensemble de chemins) les plus simples à analyser pour le processus de test.

4.1.3 Etapes de la création des bouchons

En résumé, la création automatique des bouchons pour le test structurel d'un agrégat s'effectue en quatre étapes :

1. Tout d'abord on calcule un bon ordre de parcours du graphe d'appel pour la création des bouchons pour l'agrégat (voir 4.5). Dans le cas d'un graphe d'appel acyclique, il s'agit d'un ordre ascendant. En présence de cycles dans le graphe, on appliquera une heuristique pour casser ces cycles au prix d'une création des bouchons complexifiée.
2. On produit, pour la première entité appelée de l'agrégat, son modèle (voir 4.2.3). Ce dernier est issu d'une analyse statique du code de l'entité et il est composé d'une hiérarchie d'approximations. Chaque approximation est un sous-programme obtenu par découpage du graphe de flot de contrôle de l'entité originale. Les approximations sont ordonnées par ordre croissant de complexité d'analyse.
3. Suivant le schéma retenu pour la création des bouchons, on construit ensuite les environnement d'appel de cette entité (voir 4.3.2). Un environnement complet est composé d'un contexte d'appel et d'objectifs de génération.

   Le contexte d'appel représente le domaine de définition des entrées de l'entité au point d'appel tandis que les objectifs de génération sont des contraintes sur les sorties de l'entité.
4. On produit le bouchon de l'entité appelée à partir de son modèle et de l'environnement d'appel (voir 4.3.3). On sélectionne les approximations du modèle qui seront présentes dans le bouchon spécifique à cet environnement.

   Pour les entités appelées directement depuis l'entité sous test, on construit des bouchons avec objectifs de génération. Les entités appelées de rang inférieur seront remplacées par des bouchons sans objectifs de génération.
5. On passe à l'entité suivante.

Nos bouchons suffisants pour le test structurel sont construits comme une hiérarchie d'approximations allant de la plus simple à analyser vers la plus complexe. Durant le

processus de génération de données, l'approximation placée en tête de la hiérarchie est retenue pour représenter l'entité appelée.

Si cette approximation échoue, i.e. les sorties produites par l'approximation ne permettent pas de générer une donnée de test couvrant le composant visé dans l'entité appelante, elle est retirée. Le processus de génération de données de test travaille alors à partir de l'approximation suivante de la hiérarchie jusqu'à la génération d'une donnée de test ou l'épuisement des approximations du bouchon.

Dans la suite de ce chapitre, nous présentons les techniques heuristiques que nous avons retenues de manière générer automatiquement les bouchons nécessaires au test structurel d'un agrégat.

4.1.4 Ordre de parcours du graphe d'appel pour la création des bouchons

Idéalement, lorsqu'on souhaite produire le modèle d'une entité, toutes les entités qu'elle est suceptible d'appeler doivent déjà posséder un bouchon à même de les remplacer avantageusement. On pourrait donc procéder à une création ascendante des bouchons suivant le graphe d'appel : lors de la construction du modèle d'une entité de rang n dans l'agrégat, toutes les entités de rangs inférieurs possèderaient déjà un modèle.

Cependant, une telle politique ne fontionnera pas en présence d'entités récursives ou mutuellement récursives. Déterminer un bon ordre de construction demande de proposer une méthode pour casser les cycles du graphe d'appel de l'agrégat.

A notre connaissance, le problème général posé par l'interdépendance des entités pour le test n'a pas été abordé pour les langages de programmation tels que C. Par contre, ce problème fait l'objet de recherches pour le test d'intégration de logiciel basé sur des technologies orientées objet, leur but étant de limiter le besoin de bouchons lors de l'intégration d'une classe dans le logiciel.

Dans [LCB03] on trouve une comparaison de différentes approches répondant à cet objectif. Ces approches sont basées sur un graphe représentant les dépendances entres les classes du logiciel. Elles proposent plusieurs heuristiques pour casser les interdépendances, en se basant sur les différents types de dépendances présentes dans les cycles ou sur le rang des classes dans le graphe.

Les solutions qu'elles proposent ne sont pas appliquables au calcul d'un bon ordre de construction des bouchons. En effet, elles reposent sur des relations entre entités qui n'existent pas dans un graphe d'appel. On y trouve, par exemple, les relations d'héritage entre les classes. Ce surplus d'informations permet à ces techniques de proposer une analyse fine des implications engendrées par la suppression d'un arc du graphe.

Malgré tout, leur but reste très similaire au nôtre : trouver un bon compromis entre le nombre (ou la complexité) des bouchons à créer - dans notre cas il s'agit du nombre (ou de la complexité) du corps des fonctions à analyser - et la suppression d'un maximum de cycles dans le graphe considéré.

La solution pratique que nous avons retenue pour ce problème est présentée dans la section 4.5. Dans le cas d'un graphe d'appel acyclique nous calculons un ordre de construction des bouchons ascendant sur le graphe d'appel. En présence de cycles dans le

graphe, nous nous sommes inspirés d'heuristiques proposées pour l'intégration de classes dans les logiciels à objets.

4.2 Approximations et modèle

4.2.1 Introduction

Pour préciser les notions de modèles et d'approximations, on introduit les définitions suivantes :

Définition 4.1 *On étend la définition du graphe de flot de contrôle avec la notion de nœuds d'assertion. On appelle* nœud d'assertion*, un nœud du graphe de flot de contrôle, qui est étiqueté par une contrainte. Cette contrainte s'exprime à l'aide des variables de l'entité.*

Pour traverser un nœud d'assertion, les valeurs des variables à l'exécution doivent vérifier la contrainte.

Définition 4.2 Déterminer *un nœud de décision* n *d'un graphe de flot de contrôle consiste à choisir un arc* a *parmi tous ceux qui ont leur origine en* n*. Les arcs non sélectionnés sont supprimés et on transforme* n *en un nœud d'assertion.*

La contrainte de ce nœud est construite à partir de la décision qui étiquette n*. Cette contrainte est l'expression dont la satisfaction est nécessaire au passage par* a.

Définition 4.3 *On appelle simplification d'un opérateur de contrôle (et par extension simplification d'un nœud de décision), noté* simplifie(n,a)*, l'opération de suppression de l'arc* a *issus du nœud de décision* n *associé à l'opérateur de contrôle dans le graphe de flot de contrôle de la fonction.*

On distingue deux situations après la suppression de l'arc :
- $Out(n) = 1$*, on détermine* n *pour l'arc restant.*
- $Out(n) > 1$*, cette situation peut uniquement se produire dans le cas d'un opérateur de contrôle de type switch qui repose sur de multiples décisions. Dans ce cas, le type de* n *n'est pas modifié. On s'est contenté de supprimer certains "case" du switch.*

Le graphe de flot de contrôle de chaque entité appelée de l'agrégat est scindé en un ensemble de sous-graphes qui constitue le *modèle* de l'entité. Un sous-graphe définit une *approximation*; il représente un sous-programme.

Une approximation peut être vue comme une tranche de l'entité à la manière du "slicing" intra-procédural [Wei84, RT96, Tip95, Luc01]. La principale différence entre une tranche et une approximation est le critère de découpe. Dans le premier cas, il s'agit de la définition d'une ou plusieurs variables à un point donné de l'entité, dans le second on souhaite obtenir un sous-programme dont l'exécution couvre obligatoirement un composant de l'entité.

Le résultat de notre découpe est un sous-programme équivalent à l'entité originale sous certaines conditions d'utilisation, représentées par les nœuds d'assertions.

Une approximation extraite du graphe de flot de contrôle de la figure 4.5 pourrait être le sous-graphe de la figure 4.6. Cette approximation représente les comportements de la

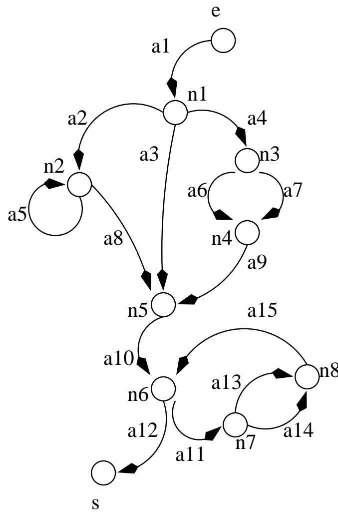

Fig. 4.5 – Un exemple de graphe de flot de contrôle

fonction originale ayant pour point commun le passage par l'arc a_3. Dans notre exemple, forcer le passage par cet arc n'entraine aucune modification sur la suite de l'exécution de la fonction (représentée par les arcs a_{10} à a_{15} et les nœuds n_5 à s).

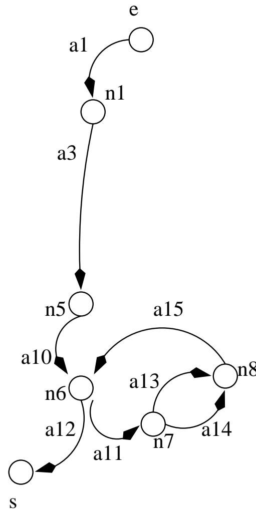

Fig. 4.6 – Une approximation possible

On note que le sous-graphe de la figure 4.6 n'est pas un graphe de flot de contrôle dans la mesure où on peut y trouver des nœuds d'assertion.

Un nœud d'assertion possède un unique arc entrant et un unique arc sortant. Pour retrouver le graphe de flot de contrôle de l'approximation, on fusionne le nœud d'assertion et ses arcs pour former un unique arc dans lequel l'assertion est intégrée comme une instruction. Le graphe de flot de contrôle de la figure 4.6 obtenu est illustré par la figure 4.7.

Finalement, le modèle d'une entité est une hiérarchie d'approximations où chaque arc accessible de l'entité est présent sur tous les chemins d'exécutions d'au moins des approximations du modèle. Autrement dit, quelle que soit l'exécution de cette approximation,

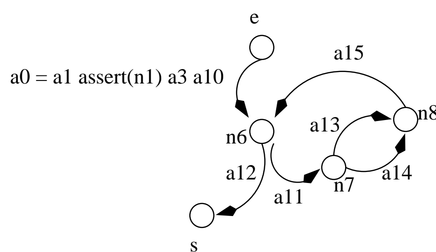

FIG. 4.7 – Le graphe de flot de contrôle de l'approximation

elle exécute cet arc.

Au sein de la hiérarchie, les approximations sont ordonnées de la plus simple à la plus compliquée.

Lorsqu'on parle de la complexité d'une approximation, on entend parler de la difficulté posée au processus de génération de données de test pour analyser son code. Cette analyse doit déterminer quelle donnée est nécessaire en entrée de l'approximation pour engendrer une sortie permettant la génération d'une donnée dont l'exécution va couvrir le composant visé de l'entité appelante.

Intuitivement, cette complexité d'une approximation prolonge celle présentée en 4.1.2 :

- La complexité d'une entité séquentielle est définie par le nombre de instructions qui la compose.
- Lorsqu'on introduit des points de choix, à travers les opérateurs de contrôle d'un langage, le nombre de chemins à analyser augmente de manière combinatoire. Une approximation contient généralement de tels points de choix ; sa complexité est alors fonction de celle de ses instructions pondérée par le nombre de nœuds de décision à traverser pour les atteindre.
- Il peut également arriver qu'une approximation contienne des appels à d'autres entités de l'agrégat. Ces dernières nécessiteront à leur tour une analyse. La complexité de cette analyse va se reflèter dans la complexité de l'approximation.

Intuitivement, les approximations regroupées dans le modèle correspondent à différents comportements possibles de l'entité originale ; il s'agit de mettre en avant ceux qui sont les plus simples à analyser de manière à ce qu'ils soient utilisés en priorité par le processus de test.

Le choix de l'ensemble des approximations à regrouper dans le modèle et la manière dont elles sont extraites de l'entité originale sont détaillés dans la suite de cette section.

En particulier, en l'absence de toute spécification des entités de l'agrégat, nous avons défini un critère d'arrêt de l'extraction des approximations d'une entité basé sur la couverture des composants de cette entité ; ce critère est similaire aux critères structurels utilisés lors du test unitaire.

4.2.2 Approximations

Nous allons construire l'approximation qui assure la couverture de l'arc a d'un graphe G. Afin d'aider la compréhension, nous accompagnons les définitions par le graphe de flot de contrôle $G = (N, A, e, s)$ illustré par la figure 4.8.

On définit les deux ensembles suivants :

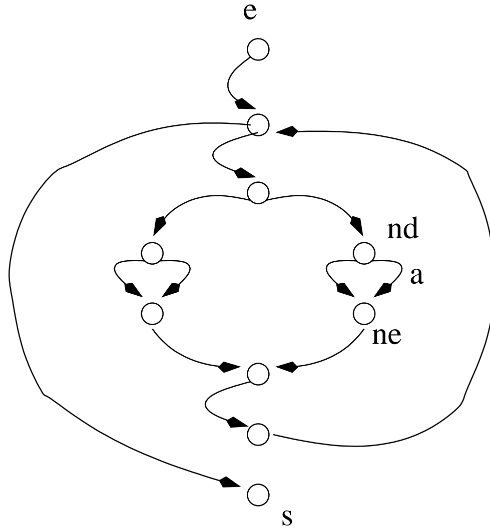

FIG. 4.8 – Un graphe de flot de contrôle

- $C_1 = \{c = a_0...a_i/origine(a_0) = e, extremite(a_i) = n_d\}$, l'ensemble des chemins issus de e et allant jusqu'à n_d, le premier nœud de décision dont dépend a en remontant le graphe de flot de contrôle.
- $C_2 = \{c = a_j...a_k/extremite(a) = n_e, origine(a_j) = n_e, extremite(a_k) = s\}$, l'ensemble des chemins issus de n_e, le nœud atteint par a, et allant jusqu'à s.

On note qu'en présence de boucle dans les fonctions, ces ensembles de chemins peuvent être infinis. En pratique, pour pallier ce problème, on limite le nombre d'itérations dans les boucles. Par exemple, on ne créera les ensembles C_1 et C_2 que pour 0 ou 1 passage dans les boucles *while*.

et l'ensemble des nœuds et des arcs qui interviennent dans ces ensembles de chemins :
- $A_1 = \{a \in A/\exists c \in C_1, c = a_0...a...a_i\}$
- $N_1 = \{n \in N/\forall a \in A_1, origine(a) = n\}$
- $A_2 = \{a \in A/\exists c \in C_2, c = a_j...a...a_k\}$
- $N_2 = \{n \in N/\forall a \in A_2, origine(a) = n\}$

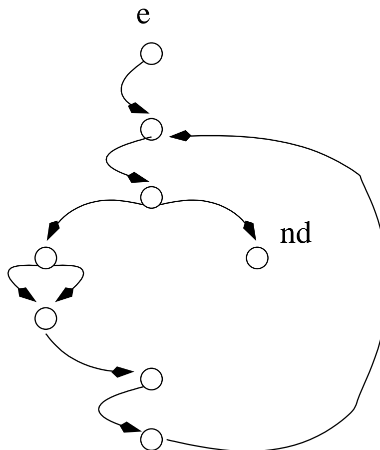

FIG. 4.9 – P_1 : le sous-graphe précédent a dans G

FIG. 4.10 – P_2 : le sous-graphe suivant a dans G

Pour aider à la compréhension du lecteur, on peut noter qu'à partir de ces ensembles, on peut définir le graphe $P_1 = (N_1, A_1, e, n_d)$ (figure 4.9) et $P_2 = (N_2, A_2, n_e, s)$ (figure 4.10). Intuitivement, il s'agit du sous-graphe qui précède (resp. suit) a dans G.

Soit $A_0 = \{a \in A / \exists n \in N_1, origine(a) = n\}$, l'ensemble de tous les arcs de G issus des nœuds de N_1.

Soit $A_{x0} = A_0 - A_1$, l'ensemble des arcs de G issus des nœuds de N_1 qui n'apparaissent pas dans P_1.

On construit maintenant l'ensemble des nœuds qui doivent être simplifié pour atteindre par a :

$$D_0 = \{n \in N_1 / \exists a \in A_{x0}, origine(a) = n\}$$

On note $Det(D_0)$ l'ensemble des nœuds simplifiés tels que :

$$\forall n_d \in D_0, \exists d \in Det(D_0) / \forall a \in A_{x0}, origine(a) = n_d, d = simplifie(n_d, a)$$

On peut maintenant modifier le type des arcs de P_1 pour refléter ces simplifications :

$$N_1'' = N_1 - D_0 \cup Det(D_0)$$

Pour effectivement couvrir l'arc a, on détermine le nœud n_d pour a. Il est transformé en un nœud d'assertion étiqueté par la contrainte $cont(n_d)$.

On construit alors l'ensemble des chemins qui satisfont cette contrainte et toutes les contraintes des nœuds d'assertion de $Det(D_0)$, c'est à dire l'ensemble des chemins d'exécution qui assurent la couverture de a :

$$C_1' = \{c \in C_1 / c\ satisfait\ cont(n_d)\ et\ \forall d \in Det(D_0), cont(d)\}$$

On définit les ensembles A_1', N_1' des arcs (resp. des nœuds) qui interviennent dans les chemins de C_1 :

- $A_1' = \{a \in A / \exists c \in C_1', c = a_0 ... a ... a_i\}$
- $N_1' = \{n \in N_1'' / \forall a \in A_1', origine(a) = n\}$

On peut noter que $S = N_1 - N'_1$ est l'ensemble des nœuds supprimés par la détermination de n_d.

Soit $A_{x_1} = A_1 - A'_1$, l'ensemble des arcs supprimés par la satisfaction de la nouvelle contrainte introduite par $cont(n_d)$.

On note $Det(D_1)$ l'ensemble des nœuds simplifiés tels que :

$$\forall n_d \in D_1, \exists d \in Det(D_1) / \forall a \in A_{x_1}, origine(a) = n_d, d = simplifie(n_d, a)$$

On construit le nouvel ensemble de nœuds :

$$N'''_1 = N''_1 - D_1 \cup Det(D_1)$$

A ce stade, nous avons traité le sous graphe P_1. En particulier, nous avons simplifié tous les nœuds de décision de P_1 qui empêchent la couverture systématique de a.

Il nous faut maintenant travailler sur les implications de ces modifications sur le sous-graphe P_2.

Dans ce but, on construit C'_2 l'ensemble des chemins c_1, a, c_2 tel que $c_1 \in C'_1$ et $c_2 \in C_2$.

On élimine ensuite de cet ensemble, les chemins qui ne satisfont pas les assertions des nœuds simplifiés issus de D_1 :

$$C''_2 = \{c \in C'_2 / \forall n \in N'''_1 et\ type(n) = assertion, c\ satisfait\ cont(d)\}$$

On définit ensuite les ensembles A'_2, N'_2 des arcs (resp. des nœuds) qui interviennent dans les chemins de C''_2. On ajoute le nœuds de sortie à N'_2 :

- $A'_2 = \{a \in A / \exists c \in C''_2, c = a_0 ... a ... a_i\}$
- $N'_2 = \{n \in N / \forall a \in A'_2, origine(a) = n\} \cup s$

Comme précédement, on détermine l'ensemble des nœuds à simplifier dans le sous-graphe qui suit a dans G en construisant l'ensemble des arcs éliminés A_{x_2} :

$$A''_2 = \{a \in A / \exists n \in N'_2, origine(a) = n\}$$
$$A_{x_2} = A''_2 - A'_2$$

Puis les nœuds à l'origine de ces arcs qui n'ont pas déjà été simplifié précédement :

$$D_2 = \{n \in N'_2 / \exists a \in A_{x_2}, origine(a) = n\ et\ type(n) \neq assertion\}$$

On note $Det(D_2)$ l'ensemble des nœuds simplifiés tels que :

$$\forall n_d \in D_2, \exists d \in Det(D_2) / \forall a \in A_{x_2}, origine(a) = n_d, d = simplifie(n_d, a)$$

On peut finalement construire l'ensemble des nœuds de l'approximation comme :

$$N''_2 = N'_2 - D_2 \cup Det(D_2)$$

Finalement :

Définition 4.4 *Une approximation $A(a)$ issue du graphe de flot de contrôle $G = (N, A, e, s)$ avec $a \in A$ est le sous-graphe $G' = (N''_2, A'_2, e, s)$.*

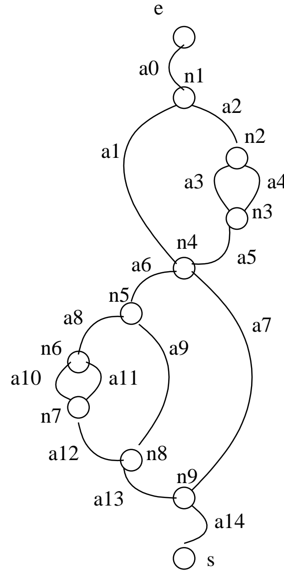

Fig. 4.11 – Différentes imbrications des arcs d'un GFC

4.2.3 Modèle

Choix des arcs et critère d'arrêt

En dehors de toute spécification fonctionnelle des entités, nous définissons les approximations à partir d'un critère structurel : la couverture d'un composant. Pour une première approche, nous avons choisi de couvrir les arcs du graphe de flot de contrôle [KCG04].

Dans un premier temps, nous considérons qu'un ensemble d'approximations est suffisant pour représenter une entité lorsque tous les arcs de l'entité originale sont forcément couverts par l'exécution d'au moins une approximation du modèle.

Ce choix heuristique n'est évidemment pas satisfaisant mais d'autres choix plus à même de représenter les comportements d'une entité nécessiteraient des spécifications fonctionnelles de ces comportements [LPU04, BLUV04].

La trame de l'extraction L'algorithme d'extraction d'approximations est décomposé en deux fonctions :

- La fonction de marquage des arcs, chargée de l'arrêt de l'extraction.
- La fonction de construction des approximations *A()*.

L'algorithme 4.1 choisit le premier arc à couvrir. Ce choix est heuristique, il influe le nombre et la complexité des approximations extraites de l'entité.

Nos expérimentations semblent indiquer que le choix de l'arc le plus imbriqué dans le graphe de flot de contrôle permet d'extraire un plus petit nombre d'approximations ainsi que des approximations plus simples à analyser. Ceci s'explique par le nombre de nœuds de décisions déterminés par le choix de cet arc.

En effet, pour couvrir l'arc a_{10} du graphe de flot de contrôle de la figure 4.11, on doit nécessairement passer par les arcs a_8 et a_6. On détermine donc les décisions des nœuds n_4 et n_5 en même temps que celle du nœud n_6 dont a_{10} dépend.

Si on avait, par exemple, choisi de parcourir les arcs par ordre d'apparition dans le graphe, l'approximation couvrant l'arc a_1 (a_0 est couvert par toutes les approximations) ne détermine pas d'autres nœuds de décision que n_1 dont il dépend.

De plus, notre critère de complexité, détaillé plus loin dans ce chapitre, dépend fortement du nombre de nœuds de décisions présents dans les approximations. Plus on en détermine, et par là même plus on se rapproche d'une approximation séquentielle, plus l'approximation est considérée comme simple à analyser.

★ **Algorithme 4.1** : *Trame de l'extraction*

do
  ▶ *Trouver l'arc* a_i *le plus imbrique*
  ▶ *Extraction de l'approximation* $A(a_i)$
  ▶ *Marquer tous les arcs couverts par* $A(a_i)$
while *Tous les arcs ne sont pas couverts*

Marquer les arcs La condition d'arrêt de l'algorithme est l'obtention d'une couverture complète des arcs de l'entité.

Lorsqu'une approximation est extraite, l'algorithme 4.1 marque les arcs non imbriqués qui la compose. On ne marque que les arcs du graphe de flot de contrôle de l'entité qui sont nécessairement couverts par l'exécution de l'approximation.

Par exemple, l'exécution de l'approximation $A(a_3)$ de la figure 4.6 couvre nécessairement les arcs a_1, a_3, a_{10} et a_{12}. Bien que les arcs a_{11}, a_{13}, a_{14} et a_{15} soient présents dans l'approximation, on ne peut pas garantir qu'ils seront couverts par toutes les exécutions de $A(a_3)$. Le choix de ne marquer aucun de ces derniers nous assure que l'algorithme calculera une autre approximation spécifique à ces arcs pour le modèle.

L'algorithme ne pourra pas extraire une approximation $A(a)$ d'un graphe de flot de contrôle G lorsque :

- l'arc a n'est pas atteignable dans G, nous verrons dans la partie II que notre implémentation peut détecter cette situation ;
- le calcul de l'approximation ne termine pas dans le temps qui lui était imparti. Dans ce cas, l'arc a sera tout de même marqué. L'algorithme a essayé de calculer $A(a)$, il a échoué. Cet arc ne sera pas présent dans le modèle.
  Ce comportement peut avoir un impact sur la génération de donnée. En effet, rien ne garantit que l'arc en question est non atteignable. Pourtant, les comportements qui lui sont liés ne seront pas présents dans le modèle.

Dans ces deux cas, les arcs en questions sont considérés comme non atteignables.

A la fin de l'exécution de cet algorithme, on a calculé un ensemble d'approximations couvrant tous les arcs considérés comme atteignables du graphe de flot de contrôle de l'entité originale.

La fonction d'extraction La fonction d'extraction a pour rôle de calculer une approximation $A(a)$. La définition d'une approximation donnée en 4.2.2 se base sur la

construction d'ensembles de chemins. Toutefois, il est important de noter qu'en pratique, la construction de ces ensembles n'est pas effectuée. En effet, le contexte de cette thèse, en particulier l'utilisation de la programmation logique avec contraintes, permet de construire une approximation uniquement par propagation de l'assertion $cont(n_d)$. Nous verrons comment plus en détails dans le chapitre 7.

Complexité des approximations

Pour couvrir un composant c de l'entité f et dépendant d'un appel à l'entité g, le processus de test peut indifféremment utiliser un comportement complexe ou simple de g ; seule la valeur produite est importante. Pour les besoins de la couverture structurelle, le comportement simple de g facilite le test.

Pour appliquer cette préférence à la simplicité, nous avons défini un ordre sur les approximations basé sur les instructions qui les composent.

Le coût associé à chaque instruction dépend du processus de génération de données effectivement utilisé. Par exemple, on peut imaginer que le coût de l'analyse d'une division n'est pas le même que celui d'une addition pour un processus de génération. Cette différence peut être accentuée par les types de données utilisées, les calculs sur les flottant étant plus complexes que ceux réalisés sur les entiers.

Lorsqu'on a associé un poids à chaque instruction du langage considéré, on peut calculer le poids d'une entité ou de ses approximations. Le poids d'une approximation donne alors une indication de la difficulté que son analyse posera au processus de génération.

Le poids global d'une entité n'est pas normée. Notre heuristique pondère le graphe de flot de contrôle de chaque entité.

En pratique, la valeur des poids est liée au processus de génération de données utilisé. Pour autant, les formules qui permettent d'obtenir ces poids reposent sur les mêmes constatations :

- Lorsqu'une entité admet des points de choix, sa complexité est d'autant plus importante que le processus de génération devra analyser chaque branche, voir chaque chemin, pour déterminer celui qui est nécessaire à produire ses données.
- L'équiprobabilité *a priori* de la nécessité d'exécuter les arcs issus d'un nœuds de décisions. Lors de l'analyse statique du corps d'une entité, aucun des arcs issus d'un tel nœuds n'a plus de chance d'être celui dont le processus de génération a besoin pour produire une donnée.

Par conséquent, ces formules de mesure suivent toute un même schéma :

- $poids(if...then...else) = poids(then) + poids(else) + CIF$: le poids d'un opérateur de contrôle de type $if...then...else$ ou $switch$ est la somme des poids de chacune de ses branches.
  On effectue la somme car, potentiellement, le processus de génération devra toutes les analyser une fois afin de déterminer ses données.
- $poids(Boucle(C)) = poids(C) * CW$: Le poids d'une boucle est donné par le poids du corps de cette boucle multiplié par une constante CW.
  L'utilisation de la multiplication par une constante se justifie pour représenter une valeur moyenne de passage dans la boucle.
- $poids(appel(f(x)) = poids(f) + CC$: Le poids de l'appel d'une entité est en rapport direct avec le poids de l'entité appelée.

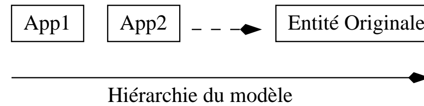

FIG. 4.12 – Architecture du modèle d'une entité

- $poids(Instruction) = 1$: ici nous avons fait le choix de considérer que toutes les instructions élémentaires du langage sont de même complexité. Ce choix est à adapter au processus de génération de données.

Des constantes sont présentes dans ces forumles. Elles servent principalement à les adapter aux processus de génération qui sera effectivement utiliser.

On note que leur valeur numérique est arbitraire mais que le rapport entre les différentes constantes est utile pour classer les opérateurs de contrôle en fonction de la difficulté d'analyse qu'ils posent au processus de génération.

Par exemple, pour Inka, le traitement d'un appel de fonction est plus complexe qu'un opérateur de boucle (si on considère que la boucle est exécutée au plus une fois), lui même plus complexe qu'un $if...then...else$.

Le poids d'une approximation est obtenu en additionnant le poids de chacun de ses arcs. Une règle similaire aux marquages des arcs couverts par le modèle d'une entité et aux formules de poids est utilisée :

- Lorsqu'un arc est nécessairement couvert par l'exécution de cette approximation, alors son poids est ajouté au poids de l'approximation.
- Lorsqu'une approximation contient un nœud de décision non déterminé, la règle spécifique au poids de ce nœud est utilisée.

Construction de la hiérarchie

On ordonne finalement les approximations par leur poids de sorte que la plus simple soit en tête du modèle. Celui-ci est alors formé d'une hiérarchie d'approximations comme schématisé par la figure 4.12.

On note que, pour répondre à la perte de comportements de l'entité originale introduite par :

- la simplification des boucles (aucun ou un seul passage) pour la construction des approximations,
- l'arrêt du calcul des approximations lorsque celui ci dépasse le temps qui lui était imparti,

on introduit, en dernière approximation du modèle, la fonction originale. De cette manière, on assure que les bouchons issus de ce modèle permettront de couvrir les mêmes composants des entités appelantes que l'aurait fait l'entité appelée.

L'ordonnancement du modèle assure que le dépliage du code complet de la fonction orginale n'a lieu qu'en tout dernier recours, lorsque toutes les autres approximations ont échouées.

4.3 Bouchons

4.3.1 Contexte d'une variable

Nous calculons un environnement pour toutes variables numériques. On définit les notions de domaines et de contextes d'une variable pour caractériser formellement un environnement :

Domaine d'une variable

Définition 4.5 *Le* domaine d_x *d'une variable x est un sur-ensemble du domaine de définition de la variable x. C'est un intervalle défini par la borne supérieure et par la borne inférieure du domaine de définition de x.*

Par exemple si le domaine de définition de x, $dom(x) = \{1, 2, 3\} \cup [6, 12]$ alors le domaine de x, $d_x = [1, 12]$.

L'évolution de l'ensemble des domaines le long d'un chemin va exercer une influence sur les chemins empruntables dans la fonction appelée. Ces modifications se manifestent par l'introduction de nouvelles contraintes sur les domaines par chaque arc.

Définition 4.6 *On appelle* l'évolution du domaine d_v par une instruction I qui définit v, *le domaine d'_v qui est la solution de l'équation sur les domaines obtenue par substitution dans I des variables par leur domaine respectif au point de cette instruction.*

Par exemple, si on note $d_v = [0, 10]$ et $I =$ "$v = 2 * y + 3$" avec $d_y = [0, 1]$ alors $d'_v = 2 * [0, 1] + 3 = [3, 5]$.

On note que chaque évolution d'un domaine d_v par une instruction I peut avoir un effet sur toutes les variables utilisées dans l'instruction car l'équation issue de I doit admettre une solution dans les bornes min, max du domaine de chaque variable.

Par exemple, si $d_y = [min, max]$ et $I =$ "$v = 2*y+3$" alors $d_y = [\frac{min}{2} + 3, \frac{max}{2} - 3]$ et $d_v = [min, max]$.

Evolution du contexte d'une variable sur un arc Le contexte d'une variable v au nœud n représente toutes les évolutions possibles de son domaine le long des chemins issus de e jusqu'au nœud d'intérêt n.

Le *contexte d'une variable* v en un nœud n, noté $C_{(v,n)}$, est défini par la liste des domaines de la variable v au nœud n. Il y a autant de domaines de v dans cette liste qu'il y a de chemins depuis e jusqu'au nœud n.

Au nœud e (origine du graphe de contrôle), toutes les variables de la fonction ont un contexte initial défini par $C_{(v,e)} = [min, max]$ où min et max sont les valeurs aux bornes du domaine défini par le type de la variable v.

Pour obtenir le contexte des variables au nœud d'intérêt n, il faut calculer toutes les évolutions possibles de $C_{(v,e)}$ sur les chemins menant à n.

L'évolution d'un contexte par un arc est définie comme :

Définition 4.7 *Le contexte $C_{(v,n)}$ est l'évolution du contexte $C_{(v,m)}$ par l'arc a lorsque :*
- *$origine(a) = m$ et $extremite(a) = n$.*

– *Les domaines de* $C_{(v,n)}$ *sont les évolutions des domaines de* $C_{(v,m)}$ *par application de toutes les instructions élémentaires de l'arc* a *qui définissent* v.

Par convention, on note alors que $C_{(v,n)} = C_{(v,m)} \dot{\cap} \overline{a_i}$ *avec* $\dot{\cap}$ *l'opérateur associatif à gauche représentant, selon le type de* m *:*

– *Si* type(m)= décision *alors l'évolution des domaines du contexte* $C_{(v,m)}$ *par la contrainte représentée par l'expression symbolique conditionnant l'exécution de* a *puis par les instructions élémentaires de l'arc* a *dans leur ordre d'apparition.*
– *Si* type(m)=appel *à une fonction* $g()$ *alors l'évolution du contexte* $C_{(v,m)}$ *par toutes les instructions de* $g()$ *à travers tous ses chemins d'exécution*[1][2] *et les éventuelles instructions élémentaires présentes sur* a *dans leur ordre d'apparition.*
– *Si* type(m)=jonction *alors l'évolution des domaines du contexte* $C_{(v,m)}$ *par les instructions élémentaires de l'arc* a *dans leur ordre d'apparition.*

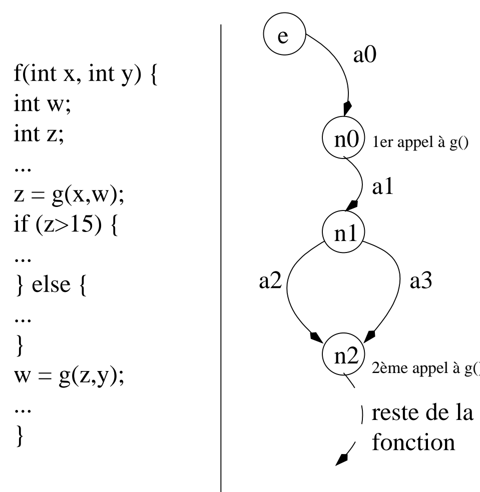

FIG. 4.13 – Deux contextes d'appels

La figure 4.13 illustre ce problème, on constate que le calcul de $C_{(w,n_0)}$ ne pose pas de difficultés. Par contre, le calcul de $C_{(z,n_2)}$ demande d'être capable de calculer l'évolution des d_z le long des chemins de n_0 à n_2. En particulier, de pouvoir calculer l'évolution de d_z sur l'arc a_1 retour du premier appel à $g()$.

Toutes les définitions précédentes sont illustrées sur l'exemple suivant :

Le graphe de flot de contrôle de la fonction f (figure 4.14) est représenté sur la figure 4.15. Nous rappelons le type des nœuds et des arcs de ce graphe (voir 2.1.2) dans les tableaux suivants :

Pour l'appel à la fonction g au nœud $n2$ de la figure 4.15 on a : $C_{(g,n2)} = C_{(w,n2)}$. Le contexte $C_{(w,n2)}$ est l'évolution du contexte de la variable w le long de tous les chemins issus de l'arc e et atteignant le nœud $n2$. En restreignant les boucles à 0 ou 1 exécution, $C_{(w,n2)}$ est l'évolution du domaine de w le long des deux chemins $e, a_0, n_0, a_1, a_2, n_1, a_3$ et $e, a_0, n_0, a_2, n_1, a_3$.

[1] On note qu'en général, le calcul de cette évolution est très complexe car $g()$ peut appeler d'autres fonctions et amener à "déplier" la totalité du graphe d'appel.

[2] En se référant à la définition formelle du contexte qui suit, cette évolution pourrait s'écrire $C_{(v,m)} \dot{\cap} C(v,s)_{g()}$.

```
int f(int x, int y) {
int w = 0;                 ]  a0
int i = x−2;               ]
while (i <= 10) {          ]  n0
  y = y * x;               ]
  i = i + 1;               ]  a1
}                          ]
w = x + y − i;             ]  a2
if (w < 0) {               ]  n1
  w = g(w);                ]  a3, n2, a4
} else {
  w = w * 10;              ]  a5
}

return w;                  ]  a6
}
```

FIG. 4.14 – Type des nœuds et des arcs de la fonction f()

	decision	jonction	appel
n0	X	X	
n1	X		
n2			X
n3		X	

TAB. 4.1 – Types des nœuds de f()

Contexte d'une variable en un nœud Le calcul du contexte d'une variable à un nœud n_j peut poser problème lorsqu'il existe au moins un chemin c de e à n_j dans lequel est présent un nœud n_i de type *appel* qui définit v, c'est à dire que le domaine de v sur c est défini par l'exécution d'une fonction appelée $g()$.

Dans ce cas de figure, lorsque l'appel qui a lieu en n_i fait évoluer le domaine de la variable v, le calcul de cette évolution à travers l'appel de fonction va nécessiter l'évaluation du corps de $g()$.

En pratique cette évaluation n'est pas envisageable. On considère donc que l'évolution à travers l'appel de fonction peut avoir n'importe quel impact et on élargit donc le contexte de v à $[min, max]$.

Finalement, on peut définir plus formellement le contexte d'une variable v en un nœud n du graphe de flot de contrôle $G = (N, a, e, s)$ par :

Définition 4.8 *Soit C l'ensemble des chemins d'exécution s de e à n dans G, $C_{(v,e)} =]min, max[$ avec min et max les bornes du domaine associé au type de la variable v.*

$\forall c \in C, c = a_0, a_1, ..., a_k$ avec $origine(a_0) = e$ et $extremite(a_k) = n$,
$C_{(v,n)_c} = C_{(v,e)} \dot{\cap} \overline{a_0} \dot{\cap} \overline{a_1} \dot{\cap} ... \dot{\cap} \overline{a_k}$

Le contexte de v en n est $C_{(v,n)} = C_{(v,n)_{c_0}}, C_{(v,n)_{c_1}}, ..., C_{(v,n)_{c_j}}$

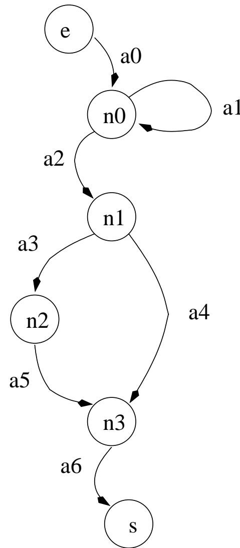

FIG. 4.15 – Graphe de flot de contrôle annoté de la fonction f()

	bloc	retour
a0	X	
a1	X	
a2	X	
a3	X	
a4	X	
a5		X
a6	X	

TAB. 4.2 – Types des arcs de f()

4.3.2 Environnement

Dans la mesure où les approximations constituant le modèle sont suffisamment simples pour être analysées efficacement, elles sont de bons candidats pour se substituer à la fonction appelée.

Comme nous l'avons vu dans la section 4.2.3, nous proposons une heuristique basée sur des poids pour comparer la complexité des approximations. Au sein du modèle, les approximations sont proposées au processus de génération de test par ordre de poids.

Toutefois, lorsque la fonction originale est découpée en un trop grand nombre d'approximations ou lorsque le nombre d'appels en cascade est important, on ne peut pas se contenter de proposer les approximations parmi toutes celles du modèle. L'environnement de l'appel courant doit nous servir comme filtre pour créer les bouchons spécifiques à partir du modèle complet de l'entité.

Imaginons le graphe d'appel de la figure 4.16. Quel est l'environnement de h ? Est-ce celui de :

- h dans g1 ?
- h dans g2 ?
- h dans g1 dans f ?

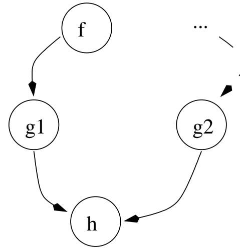

FIG. 4.16 – Quel contexte d'appel pour h ?

- h dans g1 et dans g2 ?
- h dans g1 dans f et dans g2 ?

Evidemment plus l'environnement est précis, par exemple *h dans g1 dans f et dans g2*, plus il permettra de filtrer le modèle. Pour autant, le calcul de l'environnement n'est pas un travail facile. C'est pourquoi nous avons volontairement restreint l'environnement au cas le plus simple : *h dans g1* et *h dans g2*, indépendamment l'un de l'autre.

Définition 4.9 *L'environnement d'une entité h appelée à partir de g, est noté $Cu(h)$. Il est constitué d'un contexte d'appel C_a et d'objectifs de génération Og.*

Le contexte d'appel Ca ou les objectifs de génération Og de h sont calculés dans chaque fonction appelant directement h.

Sur la figure 4.16, si on considère qu'il y a deux appels à h dans g_1 alors l'environnement de h est $Cu(h) = \{Cu(h_{g_1}), Cu(h_{g_2})\}$ et $Cu(h_{g_1}) = \{Cu(h_{g_{(1,1)}}), Cu(h_{g_{(1,2)}})\}$ où $Cu(h_{g_{(1,i)}})$ est l'environnement de h lors de son i^{ieme} appel dans g_1.

Contexte d'appel

Dans le cas où on ne crée qu'un seul bouchon par fonction, ce dernier correspond au modèle de la fonction.

Dans le cas le plus général, l'environnement nous permet de sélectionner de façon plus fine les approximations nécessaires à ce bouchon. Les approximations retenues après filtrage par l'environnement restent ordonnées.

Intuitivement, on appelle contexte d'appel d'une entité appelée dans une entité appellante, les domaines de définitions de ses entrées au noeud d'appel correspondant n_a dans le graphe de flot de contrôle de l'entité appelante.

On peut noter qu'il y a, potentiellement, pour chaque entrée, un domaine de définition par chemin de l'entité appelante entre son nœud d'entrée e et le nœud d'appel n_a. Ces domaines sont définis par le type de chaque variable et évoluent le long de tous les chemins issus de e à n_a.

Le contexte d'appel se représente naturellement pour les types de données scalaires (entiers, réels, énumération). Nous nous limitons pour le moment à ces types.

Si on reprend l'exemple de la section 3.2 :

```
int f( int x) {
1   int y;
2   int z;
3   y = 2*x*x+3;
4   z = g(y);
...
```

Le contexte d'appel de la fonction g dans f au nœud représenté par l'instruction *4* est $[3, max]$.

Dans le cas général, le contexte d'appel d'une entité est plus difficile à calculer et à représenter. Il peut exister plusieurs chemins depuis l'entrée de l'entité appelante jusqu'à l'appel à l'entité dont on souhaite calculer le contexte d'appel.

Il existe alors un domaine de définition des variables du contexte par chemin d'exécution, le contexte d'appel de l'entité est composé de tous ces domaines.

Du point de vue de g, ce contexte d'appel se présente sous la forme de contraintes portant sur son exécution à travers la restriction du domaine de ses entrées.

Définition 4.10 *Le contexte d'appel d'une entité g à un nœud n de type a de l'entité appelante, noté $Ca_{(g,n)}$, est défini par le contexte des entrées de la fonction g au nœud n.*

$$Ca_{(g,n)} = \{\forall v_i \in entree(g), C_{(v_i,n)}\}$$

Objectifs de génération

Dans un agrégat, les entités appelées se différencient par leur impact sur le flot de contrôle des entités les appelant. Ainsi, l'exécution de certains composants des entités appelantes dépend directement ou indirectement des sorties produites par l'exécution d'une ou plusieurs entités appelées.

Etant donné un composant à exécuter dans une entité appelante, on appelle *objectif de génération*, la contrainte portant sur les sorties des entités appelées dont l'exécution est nécessaire à la couverture du composant.

Intuitivement, un objectif de génération représente l'utilisation qui est faite de l'entité appelée dans l'entité appelante. Reprenons l'exemple de la section 3.2 :

```
int f( int x) {
1   int y;
2   int z;
3   y = 2*x*x+3;
4   z = g(y);
5   if (z > 2)
6      y = y + z;
7   return y;
}
```

On cherche à générer une donnée de test qui permet l'exécution du composant *6* de f (branche *then* du if then else). L'exécution de ce composant dépend du résultat de

l'appel à la fonction g en *4*. Par conséquent, l'analyse de g est nécessaire à la génération de données.

Pour atteindre *6*, la fonction g doit produire une sortie > 2. Enfin on peut noter que, dans le cas général, une même entité appelée peut avoir plusieurs objectifs de génération différents pour une entité appelante lorsque les sorties qu'elle produit interviennent dans plusieurs nœuds de décision différents.

Il est important de remarquer que toutes sorties de g supérieures à 2 sont équivalentes pour la couverture structurelle du composant *6* de f. Par conséquent, on peut définir, pour l'objectif de génération de g pour le composant c_i de f et pour permettre au processus de génération de produire une donnée assurant l'exécution de ce composant, une classe d'équivalence $\mathcal{E}q_{(g,c_i)}$ des chemins d'exécution de la fonction g.

On ne calcule pas d'objectifs de génération pour les entités dont l'exécution n'a pas d'impact sur le flot de contrôle de f. Par contre, pour celles qui en ont, on calculera autant d'objectifs de génération qu'il y a de nœuds de décision de f dont la détermination dépend du résultat de leur exécution.

Finalement, alors que le contexe d'appel pose des contraintes sur les entrées de l'entité appelée, les objectifs de génération ajoutent des contraintes sur ses sorties.

Les objectifs de génération d'un appel de $g()$ dans $f()$ sont définis par :

Définition 4.11 *Soit N_d l'ensemble des nœuds de décision de $f()$ dont l'évaluation des conditions dépend d'au moins une des sorties de $g()$.*

Pour chaque $n \in N_d$, le nombre d'objectifs de génération est égal au degré de sortie de n ($Out(n)$), et $\forall a \in A/origine(a) = n, cont(a)$ est l'expression symbolique dont la satisfaction conditionne l'exécution de a. On note S l'ensemble des sorties de $g()$ dont dépend $cont(a)$.

De plus, chaque variable a un contexte $C_{(v,n)}$ au nœud n. Soit $\forall d_v \in C_{(v,n)}, D_{(v,n)} = \bigcup d_v$, l'union des domaines de ce contexte.

On réécrit $cont(a)$ en opérant des substitutions[3] pour les variables qui sont fonctions des sorties de $g()$ par les instructions qui les définissent. Comme il peut y avoir autant de manières différentes de définir ces variables qu'il y a de chemin de e à n, $cont(a)$ devient un système d'équations-inéquations noté $cont(g()_S, a)$.

On substitue dans $cont(g()_S, a)$, chaque variable $v \notin S$ par $D_{(v,n)}$ pour obtenir le système d'équations-inéquations sur les domaines. C'est l'objectif de génération de $g()$ en a noté $Og_{(g_S,a)}$.

Les objectifs de génération pour un appel à $g()$ dans $f()$ *sont constitués de la liste des $Og_{(g_S,a)}$.*

Nous illustrons les différentes étapes du calcul d'un objectif de génération pour le premier appel à $g()$ dans $f()$ de l'exemple suivant :

[3]La réalisation des substitutions s'appuie sur des techniques de calcul symbolique

```
int f(int x) {
/*1*/   y = 2 * g(x) ;
/*2*/   z = 3*y+2 ;
/*3*/   w=12*x ;

/*4*/   if (z + w >= 0)
/*5*/     ...
        else
/*6*/     ...

        ...
}
```

Tout d'abord, sur cet exemple, il n'y a qu'un nœud de décision qui dépend du résultat de l'appel à $g() : N_d = \{4\}$. On associe à ce nœud de décision deux expressions symboliques $cont(5)$ = "$z + w \geq 0$" et $cont()$ = "$z + w < 0$" qui représentent respectivement la condition à satisfaire pour atteindre les points *5* et *6* de la fonction $f()$. Toutes deux dépendent de l'unique sortie de la fonction $g()$.

Nous allons traiter $cont(5)$, pour cela on calcule le contexte des variables et leur évolution jusqu'au nœud *4* :

$C_{(y,1)} =]min, max[$,
$C_{(y,2)} =]\frac{min}{3}, \frac{max}{3}[$, $C_{(z,2)} =]min + 2, max - 2[$,
$C_{(x,3)} =]\frac{min}{12}, \frac{max}{12}[$, $C_{(w,3)} =]min, max[$,

On effecute ensuite les substitutions dans $cont(4)$ pour obtenir $cont(g(), 5) = 6*g(x) + 12*x+2 \geq 0$ et on remplace x par l'union des domaines de son contexte au nœud *4* pour obtenir l'inéquation sur les domaines :

$$6 * g(x) +]\frac{min}{12}, \frac{max}{12}[+ 2 \geq 0$$

en prenant en admettant que x soit de type *short int* on obtient les valeurs $min = -65535$ et $max = 65535$ et l'inéquation se réécrit :

$$g(x) \geq]\frac{-65533}{72}, \frac{65533}{72}[$$

soit $g(x) \geq] - 910, 910[$. Cette inéquation constitue l'objectif de génération pour $g()$ dans $f()$ concernant *5*.

Comme pour le contexte d'appel, le calcul exact des objectifs de génération n'est pas possible lorsque le domaine d'une variable de $C_{(s,d)}$ dépend de l'exécution d'une fonction. Cela peut arriver pendant le calcul de l'évolution des contextes de variables.

Il existe également une difficulté spécifique aux objectifs de génération. Il peut arriver que les expressions symboliques fassent intervenir plusieurs appels de fonction. On pourrait, par exemple, remplacer l'instruction *3* de $f()$ par $w = 12 * h(x)$, $cont(g, 5)$ s'écrit alors $6 * g(x) + 12 * h(x) + 2 \geq 0$.

Dans ce cas, $Og(g_S, 4)$ dépend de $Og(h_S, 4)$ et inversement. On ne peut pas déterminer directement ces objectifs de génération. Une solution simple mais imprécise consiste encore à remplacer $h()$ par $]min, max[$ dans $Og(g_S, 4)$ et $g()$ par $]min, max[$ dans $Og(h_S, 4)$.

4.3.3 Deux types de bouchons

Filtrage par le contexte d'appel

Lorsqu'on détient l'environnement d'une entité ou plus généralement l'environnement d'un appel à cette entité, on se base sur le modèle de l'entité appelée pour en extraire un bouchon spécifique à cet environnement.

Dans le cadre général, comme tout bouchon correspond au moins à une entité appelée, nous considérons qu'il possède un contexte d'appel. Lorsque cela n'est pas le cas (schéma *un bouchon par entité*), on considère que ce contexte d'appel est vide, c'est à dire que le domaine de toutes les entrées est $]min, max[$.

A partir d'un modèle d'entité et d'un contexte d'appel de cette entité, on peut évaluer la pertinence des approximations du modèle. En effet, les approximations ont été construites indépendamment de tout environnement. Il est possible que le contexte d'appel courant empêche l'utilisation de certaines approximations du modèle.

```
int g( int x) {
1   int y;
2   if (x > 0)
3      y = x;
4   else
5      y = -x;
6   return y;
}
```

Soit la fonction $g()$ ci-dessus qui calcule la valeur absolue de l'entité x qu'on lui passe en paramètre. Intuitivement, le modèle de cette fonction est constitué de deux approximations, une pour chacun des arcs du graphe de flot de contrôle de g.

Pour le besoin de cet exemple, on note abusivement ces approximations *App*(3) et *App*(5), l'approximation qui assure la couverture de l'instruction *3* (resp. *5*). Le graphe d'*App*(3) contient un nœud d'assertion dont la contrainte est $x > 0$. Celui d'*App*(5) en contient un qui s'appuie sur la contrainte $x \leqslant 0$.

Suivant le contexte d'appel de g, l'utilisation de l'une ou l'autre de ces approximations peut être impossible. On distingue trois situations suivant le contexte d'appel :

- le contexte de x est inclus dans $]-min, 0]$; *App*(3) n'est pas utilisable dans ce contexte.
- le contexte de x appartient à $]0, max[$; *App*(5) n'est pas utilisable dans ce contexte.
- dans toutes les autres situations, les deux approximations peuvent être utilisées.

Une approximation est *valide* dans un contexte d'appel donné s'il y existe une exécution de l'approximation, dans ce contexte, qui satisfait ses nœuds d'assertion.

Définition 4.12 *Soit C est l'ensemble des chemins d'exécution de App, V l'ensemble de ses variables (définies ou utilisées).*

On prend comme domaine initial des entrées de App, le contexte du paramètre effectif correspondant au nœud n d'appel à $g()$ dans $f()$ noté $Ca_{(v,e)}$.

L'approximation App de $g()$ est valide dans son contexte d'appel $Ca_{(g,n)}$ dans $f()$ lorsque :

$$\exists c \in C/c = a_0, a_1, ..., a_n, origine(a_0) = e, extremite(a_n) = s/$$
$$\forall v \in V, \exists d_v \in C_{(v,s)} \text{ avec } d_v \neq \phi$$
$$\text{et } C_{(v,s)} = Ca_{(v,e)} \dot{\cap} a_0 \dot{\cap} a_1 \dot{\cap} ... \dot{\cap} a_n$$

Définition 4.13 *Un* bouchon sans objectif de génération *est construit à partir du modèle. Il est une hiérarchie d'approximations valides du modèle dans le contexte d'appel.*

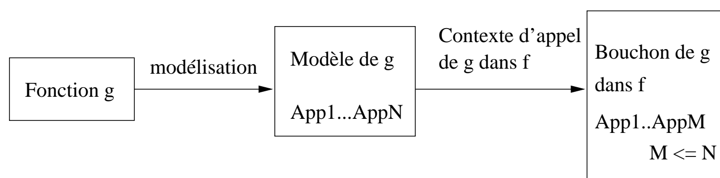

FIG. 4.17 – Utilisation du contexte d'appel pour les approximation d'un modèle

Pour construire le bouchon sans objectif de génération de l'entité g dans un contexte d'appel, on l'utilise comme un filtre pour les approximations du modèle. Le contexte d'appel nous permet d'éliminer les approximations du modèle qui ne sont pas valides.

Le contexte d'appel est constitué, pour chaque entrée de g, d'une liste de domaines. Pour obtenir un filtre, on doit combiner ces domaines. De plus, on va chercher à obtenir le filtre le plus efficace possible pour les approximations de g. Un tel filtre produira un bouchon plus simple car composé d'un nombre d'approximations moindre.

On note que, bien que considérer les domaines de chaque variable indépendament les uns des autres soit possible, cela multiplierait le nombre de bouchons créés (un par chemin de la fonction appelante jusqu'au nœud d'appel).

Pour combiner les domaines, deux choix s'offrent à nous :

- Considérer l'union des domaines pour filtrer le modèle. Le contexte d'appel considère toutes les exécutions possibles de l'entité appelante.
- Considérer l'intersection des domaines pour réduire le domaine des entrées *de l'entité appelante*. Le contexte d'appel $Ca_{(g,n_a)}$ est le plus petit contexte tel qu'il existe une valuation des entrées de l'entité appelante f exécutant g quelque soit le chemin parcouru de e à n_a dans f.

Soit la fonction f :

```
1  int f(int x)
2   if (x >= 0)
3      ...
4   else
5      ...
6  z=g(x);
7  ...
8
```

Le contexte d'appel de g dans f en *6* est $Ca_{(g,6)} = \{\{x \in (]min, -1]; [0, max[)\}\}$. Il est constitué de deux domaines pour x, chacun correspondant à une branche du $if...then..else$ de f.

On utilisera l'intersection des domaines du contexte d'appel pour filtrer le modèle lorsqu'elle n'est pas vide et l'union de ces domaines sinon.

Le filtrage s'effectue ensuite par propagation de cette intersection (ou de cette union) dans l'approximation. Elle fera partie du bouchon seulement si elle est valide dans ce contexte. On n'oubliera pas de supprimer de l'approximation valide les arcs qui ne sont pas exécutables dans le contexte courant. Cette utilisation du contexte d'appel comme filtre du modèle est illustré sur la figure 4.17.

Sélection par les objectifs de génération

Les entités appelées dont l'exécution peut influencer le flot de contrôle de l'entité appelante possèdent un ou plusieurs objectifs de génération. Pour un environnement donné, nous définissons :

Définition 4.14 *Soit App une approximation de* $g()$*. Soit* $Ca_{(g,n)}$ *le contexte d'appel de* $g()$ *au nœud n de* $f()$*. Soit C l'ensemble des chemins d'exécution de App dans* $Ca_{(g,n)}$*.*

On dit que App satisfait l'objectif de génération $Og_{(g_S,a)}$ *lorsque :*

$$\exists c \in C, c = a_0, a_1, ..., a_n, origine(a_0) = e, extremiste(a_n) = s$$
$$\forall v \in S, \exists d_v \in C_{(v,s)} / C_{(v,s)} = C_{(v,e)} \dot{\cap} a_0 \dot{\cap} ... \dot{\cap} a_n$$

et que ces d_v *sont solutions d'au moins une équation-inéquation*[4] *de* $Og_{(g_S,a)}$*.*

Définition 4.15 *Un* bouchon avec objectifs de génération *de* $g()$ *pour l'appel au nœud n de* $f()$ *est créé par sélection d'au plus une approximation du bouchon sans objectif de génération de* $g()$ *en n pour chaque objectif de génération.*

Chaque approximation sélectionnée doit satisfaire au moins un des objectifs de génération de $g()$ *en n. Une approximation peut satisfaire plusieurs objectifs de génération.*

On note que dans un tel bouchon, il y aura au maximum une approximation par objectif de génération. Il peut y en avoir moins lorsqu'une même approximation satisfait plusieurs de ces objectifs.

Les entités appelées directement par l'entité sous test, noté entité de rang 1 sur la figure 4.18, peuvent avoir un ou plusieurs objectifs de génération dans l'entité appelante (de rang 0).

Les entités appelées de rang 1 ne possèdent pas d'objectifs de génération. En effet, les entités de rang 1 n'ont pas de composants spécifiques à couvrir dans les entités de rang immédiatement inférieur.

Ces situations sont illustrées sur la figure 4.18. La procédure sous test f est de rang 0. Les procédures qu'elle appelle directement, g_1 et g_2, sont de rang 1. Leur environnement dans f contient au moins un objectif de génération, leur bouchon est donc composé d'au moins de deux approximations *AppV* et *AppF* sélectionnées pour satisfaire ces objectifs.

[4] Etre solution d'une équation-inéquation définit un chemin de $f()$ issu de e et allant jusqu'à $origine(a)$

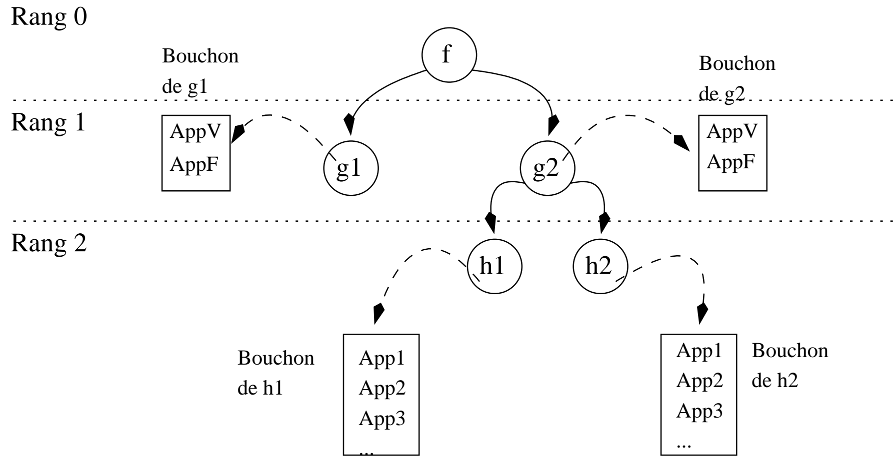

FIG. 4.18 – Deux types de bouchons selon le rang de la fonction appelée

Dans cet exemple, les approximations retenues pour le bouchon de g_2 contiennent des appels vers h_1 ou h_2, de rang 2. On ne calcule pas d'objectifs de génération pour ces h_i dans g_2. Les bouchons des h_i seront sans objectif de génération. Ils sont alors constitués d'une hiérarchie d'approximations *valides* pour les contextes d'appel des h_i dans $g2$.

Les objectifs de génération d'une entités appelées vont permettre de produire, à partir de son *bouchon sans objectif de génération*, un bouchon avec des objectifs. Ce nouveau filtrage a pour but de limiter au maximum le nombre d'approximations des bouchons.

On ne conservera, dans un bouchon avec objectifs de génération, que deux approximations pour chaque objectif de génération. Ces approximations sont les représentantes de la classe d'équivalence des sorties de l'entité pour le composant de l'entité appelante à couvrir. Si on note Og_i un objectif de génération, on ne conservera qu'une approximation valide pour $0g_i$ et une autre pour $\neg Og_i$.

Dans un bouchon avec objectifs de génération, il y aura au maximum deux approximations par objectifs de génération mais il peut y en avoir moins. En effet, lorsque l'environnement d'une entité appelée possède plusieurs objectifs de génération dans une entité appelante, il peut arriver que des approximations du modèle satisfassent plusieurs objectifs de génération de l'environnement.

4.4 Fenêtre d'utilisation des bouchons réalistes

4.4.1 Les combinaisons d'approximations

Lorsqu'on a construit, pour chaque entité de l'agrégat, son modèle, ses bouchons sans objectif de génération ainsi que ses bouchons avec objectifs de génération, les entrées du processus de test sont prêtes.

L'utilisation effective de ces bouchons pour la génération de données de test consiste à proposer, pour chaque appel dans l'entité sous test une approximation de l'entité appelée présente dans le bouchon avec objectifs de génération.

La figure 4.19 propose un exemple de système sous test modifié sur lequel on a reporté les bouchons correspondant aux appels des fonctions $g(), h(), foo1(), foo2()$.

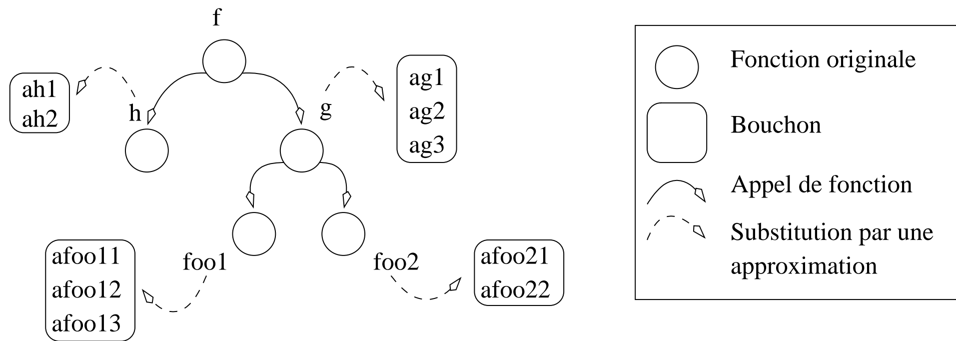

FIG. 4.19 – Entree du processus de génération de données de test

L'utilisation des bouchons réalistes pour la génération de données de test couvrant la fonction $f()$ consiste à proposer au processus de test, une combinaison suffisante des approximations de $g(), h(), foo1(), foo2()$.

Imaginons qu'un composant c de la fonction $f()$ dépende, pour son exécution, du résultat fourni par un appel à la fonction $g()$. Cet appel sera remplacé par un appel à la première approximation $ag1()$ du bouchon de $g()$.

Les approximations $ag_i()$ de $g()$ font appel à $foo1()$ et $foo2()$. Pour pouvoir être capable de produire une donnée permettant d'exécuter c de $f()$, l'analyse des $ag_i()$ doit passer par une analyse de $foo1()$ et $foo2()$. On dispose des approximations $afoo1_i()$ et $afoo2_i()$ qui remplace avantageusement $foo1()$ et $foo2()$. On propose, pour chacune de ses fonctions, une de ses approximations.

Finalement, l'analyse de l'appel à $g()$ dans $f()$ pour exécuter c consiste à analyser l'ensemble des instructions formé par les approximations $ag1(); afoo11(); afoo21()$.

Si cette combinaison d'approximation ne permet pas de produire une donnée nécessaire à la couverture de c dans $f()$, on retire la dernière approximation proposée et on la remplace par la suivante dans son bouchon. On obtient alors la combinaison :

$$ag1(); afoo11(); afoo22()$$

Sur cet exemple il y a $3 * 3 * 2$ combinaisons d'approximations à analyser dans le pire des cas. Plus l'environnement d'un bouchon permet de le spécialiser en supprimant des approximations du modèle, moins cette combinatoire est élevée. Malgré tout, dans le cadre général, on ne peut pas supposer que l'environnement seul permettra de conserver un nombre de combinaisons des approximations suffisamment petit.

4.4.2 Limiter les combinaisons : une fenêtre d'utilisation

Pour minimiser ce nombre de combinaisons nous avons, dans un premier temps, mis en place une solution technique. Comme pour la construction de modèle basé sur une limitation de l'analyse en profondeur du graphe d'appel de la section 7.2.2, nous avons définit une fenêtre d'utilisation des bouchons sur le graphe d'appel.

Dans cette fenêtre, on substitue les fonctions appelées par une approximation de leur bouchon selon le même principe que précédement. Lorsqu'on sort de cette fenêtre,

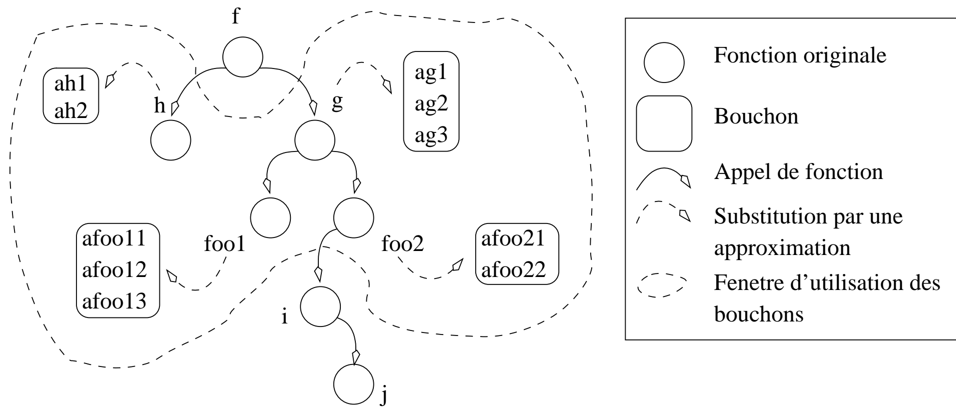

FIG. 4.20 – Une fenêtre d'utilisation des bouchons

un appel de fonction entraine obligatoirement l'analyse de toutes les instructions de la fonction.

La figure 4.20 présente une fenêtre de taille 2. C'est à dire qu'à partir de la fonction actuellement sous test, les appels de fonctions seront remplacées par des bouchons sur une profondeur de deux appels le long du graphe d'appel du système sous test.

Ici seules les fonctions $h(), g(), foo1(), foo2()$ seront bouchonnées. Si une des approximations de $foo2()$ fait appel à $i()$, le processus de test aura à analyser la totalité du corps de $i()$ et par conséquent de $j()$.

La taille de la fenêtre peut être fixé à partir du nombres d'approximations qui composent les bouchons. A chaque fois qu'on descend d'un rang de le graphe d'appel du système sous test, on est en mesure de calculer le nombre de combinaisons à analyser dans le pire des cas. De plus chaque approximations ayant un poids, on peut estimer la complexité de la fenêtre à traiter.

4.5 Déterminer l'ordre de parcours du graphe d'appel pour la construction des bouchons

Quelle que soit la fonction dont on cherche à assurer la couverture dans un graphe de flot de contrôle, toutes les fonctions auxquelles elle fait appel ainsi que celles appelées par ces dernières etc. peuvent être avantageusement remplacées par un bouchon calculé antérieurement. Intuitivement si on cherche à produire des données de test assurant la couverture structurelle de la fonction f de la figure 4.21, alors un ordre d'intégration optimal assure que g_1, g_2, g_3 ont déjà été bouchonnées ainsi que toutes les fonctions appelées par g_1, g_2, g_3. Il en découle de même que le test structurel de g_1 impose d'avoir préalablement calculé le ou les bouchons de h_1, de foo, de g_3 et de h_2.

La constructions des bouchons d'une fonction faisant appel à d'autres fonctions peut s'effectuer de deux manières. Si la fonction à simplifier appelle une fonction déjà bouchonnée, on peut choisir si on souhaite utiliser ou non les bouchons de cette fonction. Dès lors et pour maximiser l'utilisation de ces bouchons, il est nécessaire de définir un ordre de modélisation pour les fonctions à simplifier. Cet ordre cherche à maximiser le critère « les fonctions appelées dans la fonction à simplifier possèdent déjà des bouchons».

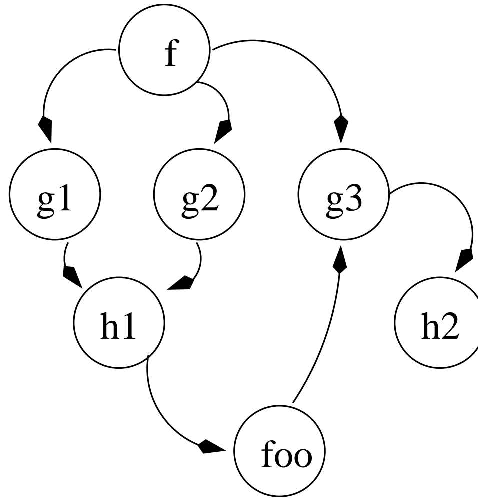

Fig. 4.21 – Un graphe d'appel

Ce critère n'est pas satisfaisable en présence de fonctions récursives ou mutuellement récursives. Le problème a donc été séparé en deux : le cas simple (aucune fonction récursive) est traité par un algorithme exact qui est ensuite étendu par une heuristique basée sur une analyse du graphe d'appel pour traiter les systèmes possédant des fonctions récursives.

4.5.1 Ordre d'intégration sans cycle

L'algorithme travaille sur le graphe d'appel du système sous test. Il construit une liste ordonnée des fonctions, qui représente l'ordre suivant lequel on construira les bouchons de ces fonctions. Il procède de la manière suivante :

- Initialisation de la liste avec les fonctions aux feuillesdu graphe d'appel (fonctions n'en appelant aucune autre).
- Ajout en fin de liste de toutes les fonctions « intégrables » à cette étape. Une fonction est intégrable lorsqu'elle n'appelle que des fonctions déjà présentes dans la liste.
- Répétition de l'étape précédente jusqu'à ce que toutes les fonctions soient intégrées à la liste.

Ceci est illustré sur la figure 4.22.

4.5.2 Ordre d'intégration avec cycle

L'algorithme simple précédent construit la liste ordonnée en supposant qu'à un instant donné il existe toujours au moins une fonction intégrable. Or, dans le cas d'un graphe d'appel comportant un ou plusieurs cycles cà n'est pas le cas. En effet, les fonctions intervenant dans ce cycle ne seront jamais intégrables car, s'appelant les unes les autres, elles ne satisferont jamais le critère d'intégrabilité tel qu'il a été défini dans la deuxième étape de l'algorithme.

Nous proposons une extension heuristique de l'algorithme. On sélectionne une fonction non intégrable choisie de façon à briser le cycle auquel elle participe : le critère de

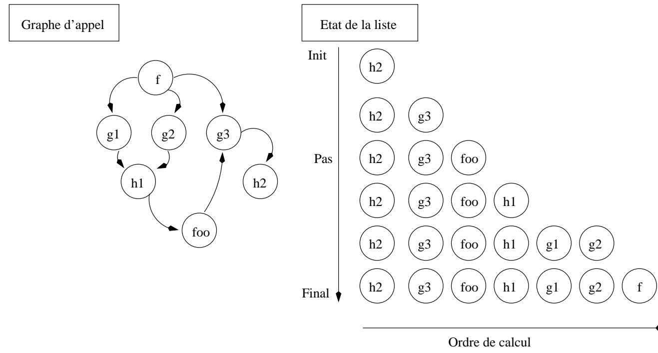

FIG. 4.22 – Un ordre de calcul des bouchons

sélection de cette fonction est quelconque. Ce principe garantit à lui seul la terminaison de l'algorithme. En effet la liste résultant de cet algorithme définit l'ordre de modélisation des fonctions de manière à assurer une utilisation maximale des modèles de fonctions déjà simplifiées. On peut toutefois modéliser une fonction sans avoir modélisé toutes les fonctions qu'elle appelle (en utilisant la véritable fonction appelée ou un bouchon classique). Dans le pire des cas, l'algorithme aboutira à un ordre d'intégration complètement arbitraire.

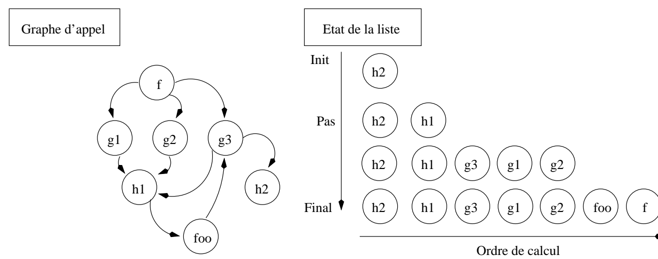

FIG. 4.23 – Un ordre possible en présence de cycles

Nous retenons comme critère de sélection d'une fonction pour briser une cycle le nombre d'appel et nous choisissons celle qui est le plus appelée. Pour déterminer ce nombre d'appel, on compte le nombre de fonctions différentes qui appelle chacune des fonctions du cycle. En effet, notre but est d'amener le plus grand nombre de fonctions a être intégrable, si une fonction f appelle plusieurs fois la fonction g présente dans un cycle, on ne comptera qu'un appel.

Le choix de la fonction ajoutée à la liste nous permet de nous approcher d'un ordre optimal d'intégration des fonctions. On choisit la fonction de manière à couper un nombre maximal de cycles potentiels dans le graphe d'appel. Pour cela on ajoute la fonction non intégrable la plus appelée par les autres fonctions non intégrables. De cette manière on augmente le nombre de fonctions potentiellement intégrables à l'instant suivant.

L'ordre produit par cet algorithme n'est pas un ordre optimal parce qu'on ne peut pas garantir que le choix de la fonction intégrée « de force » était le meilleur pour notre critère. Sur l'exemple de la figure 4.24, l'algorithme ne peut pas déterminer quelle fonction, de f ou de g, il était plus judicieux de modéliser en premier.

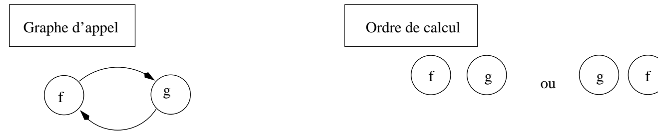

FIG. 4.24 – Un ordre arbitraire

4.6 Conclusion

Durant la génération automatique de données de test structurel pour la fonction appelante f, quand un objectif de test O_g dépend d'un appel à la fonction g modélisé, on utilise la première approximation $App1$ à la place de g pour essayer de générer une donnée d (de g) permettant d'atteindre O_g dans f. Si l'approximation proposée n'est pas adéquate (le sous-graphe ne contient pas de chemins permettant la génération de d), l'approximation est retirée et la suivante est proposée jusqu'à ce que l'objectif de test O_g puisse être atteint. Si toutes les approximations du bouchon échouent, la fonction originale complète est introduite. Ce cas de figure, bien que coûteux, assure que notre méthode sera capable d'atteindre les mêmes taux de couverture que la méthode utilisant les fonctions originales lors des appels. Ce comportement est synthétisé sur la figure 4.25.

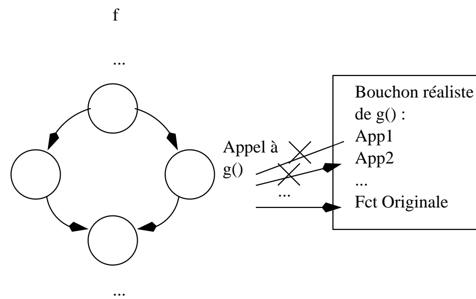

FIG. 4.25 – Utilisation d'un bouchon réaliste

Les problèmes et les questions que nous avons rencontrés lors de la conception des bouchons réalistes ne sont pas complétement nouveaux. Pour la majeure partie d'entre eux il s'agit de problème connu appliqué à un contexte différent. En particulier le concept des approximations de fonction est très largement inspiré de techniques de transformation et d'approximation et en particulier de la technique de Slicing proposé originalement par Mark Weiser [Wei84].

De manière générale ces techniques connues n'ont pas été appliquées directement à notre problème. Nous avons proposé une adatation de leur grand principe pour créer notre bouchon. Ainsi le critère de découpe du Slicing consiste à produire une représentation simplifiée de la fonction dans laquelle on a supprimé toutes les instructions qui n'ont pas d'impact sur la valeur d'une variable en un point donné de cette fonction. Dans la construction d'un bouchon ce critère n'a pas réellement lieu d'être, on va découper la fonction pour qu'elle représente l'ensemble des comportements originaux qui activent un composant de contrôle de l'entité, en l'occurence une branche du graphe de flot de contrôle.

Nous n'avons pas appliqué la technique du Slicing telle qu'elle est présenté dans

[Wei84]. C'est le contexte particulier de notre travail et en particulier l'utilisation de la PLC qui nous a permis de nous affranchir d'une implantation complète de cette technique. La traduction complète des entités sous test et les contraintes globales associés aux opérateurs de contrôle, nous permettent de calculer une approximation de l'entité originale par ajout de nouvelles contraintes portant sur le flot de contrôle - notre critère de découpe - et par résolution de l'ensemble des contraintes.

Chapitre 5

Illustration sur un exemple

5.1 Présentation de l'exemple

Afin d'illustrer le fonctionnement global de l'approche proposée dans cette thèse, nous allons dérouler ses différentes étapes sur un exemple simple.

Il est composé de quatre fonctions écrites en $pseudo - C$ qui représentent une caisse enregistreuse de supermarché. Dans la suite de cette section, nous détaillons le code des différentes fonctions ainsi que leur graphe de flot de contrôle et le graphe d'appel qu'elles forment.

```
int lenet(unsigned short ttc, unsigned short rem) {
        int net  ;
        int apayer  ;
        int pourboire ;

        /* Cette fonction ne fait appel à aucune autre */

/* 1 */  pourboire = 2 * ttc / 100 ;

/* 2 */  net = ttc - rem ;

/* 3 */  if (net <= 0)
/* 4 */          apayer = 0 ;
/* 5 */  else
/* 6 */          apayer = net + pourboire ;

/* 7 */  return apayer ;
}
```

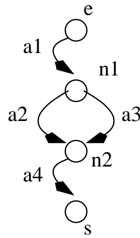

FIG. 5.1 – Graphe de flot de contrôle de la fonction lenet()

```
int APayer(unsigned short ht, unsigned short modePaiement)
{

        unsigned short ttc, apayer, rem ;

/* 1 */  rem = 0 ;

/* 2 */  ttc = ht + 120 ;

/* 3 */  if ((ttc >= 5000) && (ttc < 7500)) {
/* 4 */          if (modePaiement == 0)
/* 5 */                  rem = 700 ;
/* 6 */          else
/* 7 */                  rem = 500 ;
        }
/* 8 */  if (ttc >= 7500) {
/* 9 */          if (modePaiement == 0)
/* 10 */                         rem = 1000 ;
/* 11 */                 else
/* 12 */                         rem = 0 ;
        }
/* 13 */ if ((ttc < 5000) && (ttc > 2500)) {
/* 14 */         if (modePaiement == 0)
/* 15 */                         rem = 120 ;
/* 16 */                 else
/* 17 */                         rem = 200 ;
        }
/* 18 */ if (ttc <= 2500) {
/* 19 */         if (modePaiement == 0)
/* 20 */                         rem = 0 ;
/* 21 */                 else
/* 22 */                         rem = 5 ;
        }

/* 23 */         apayer = lenet(ttc, rem) ;

/* 24 */ return apayer ;

}
```

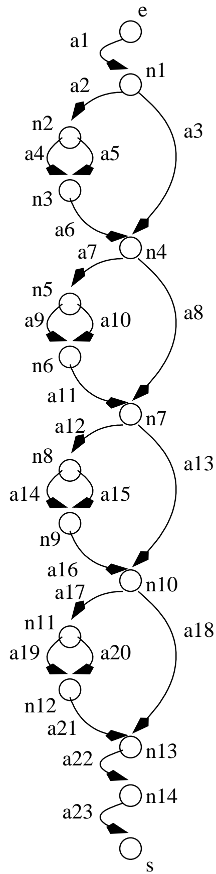

Fig. 5.2 – Graphe de flot de contrôle de la fonction Apayer()

```
unsigned short prixEURO(unsigned short somme) {
        unsigned short res=0;
        unsigned short tmp;

/* 1 */  tmp = somme + FRAIS + BENEF;

        /* Effet du passage à l'euro :-)
        Cette fonction ne fait appel à aucune autre */

/* 2 */  switch (tmp) {
/* 3 */          case (tmp <= 500)  :
/* 4 */                  res = 700  ;
                        break;
/* 5 */          case (tmp <= 700)  :
/* 6 */                  res = 900  ;
                        break;
/* 7 */          case (tmp <= 1000)  :
/* 8 */                  res = 1000  ;
                        break;
/* 9 */          case (tmp <= 1200)  :
/* 10 */                 res = 1500  ;
                        break;
/* 11 */         case (tmp <= 1500)  :
/* 12 */                 res = 1700  ;
                        break;
/* 13 */         case (tmp <= 2000)  :
/* 14 */                 res = 2200  ;
                        break;
/* 15 */         case (tmp <= 5000)  :
/* 16 */                 res = 6000  ;
                        break;
/* 17 */         case (tmp <= 10000)  :
/* 18 */                 res = 12000  ;
                        break;
/* 19 */         case (tmp > 10000)  :
/* 20 */                 res = 50000  ;
                        break;
        }

/* 21 */ return res;

}
```

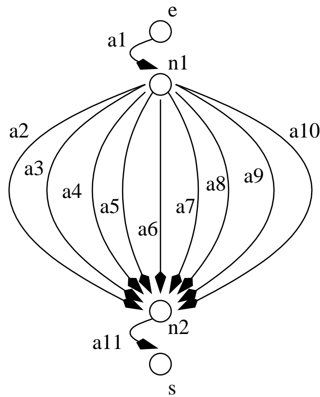

FIG. 5.3 – Graphe de flot de contrôle de la fonction prixEuro()

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FRAIS 300
#define BENEF 100


unsigned short caisse(unsigned short somme, unsigned short mode) {

        int prix;
        unsigned short liquide = 1;
        unsigned short paiement;

/* 1 */  unsigned short ht = prixEURO(somme);

/* 2 */  if ((ht > 0) && (ht <= 1000)) {
/* 3 */          prix = APayer(ht, liquide);
/* 4 */  } else if ((ht > 1000) && (ht <= 10000)) {
/* 5 */          prix = APayer(ht, mode);
        }

/* 6 */  switch (prix) {
/* 7 */          case (prix <= 6000)  :
/* 8 */                  paiement = 1; // Paiement en 1 fois
                        break :
/* 9 */          case (prix > 6000)  :
/* 10 */                 paiement = 3; // Paiement en 3 fois sans frais
                        break;
        }

/* 11 */ printf("Vous devez payer  : %d\n", prix);

/* 12 */ return paiement;
}
```

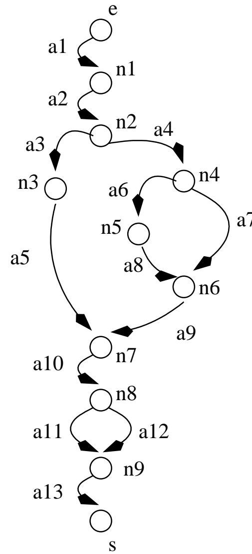

FIG. 5.4 – Graphe de flot de contrôle de la fonction caisse()

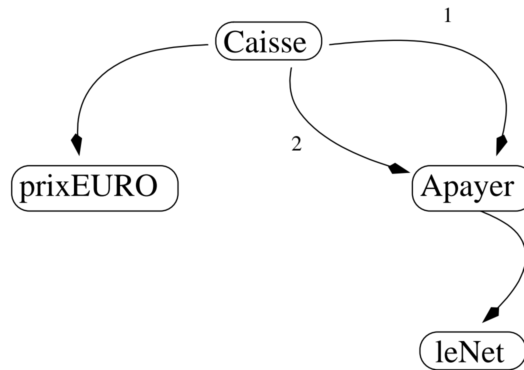

FIG. 5.5 – Le graphe d'appel de notre exemple

Ces fonctions s'appellent les unes les autres et forment le graphe d'appel de la figure 5.5.

5.2 Construction des modèles

La première étape du travail consiste à construire le modèle de chacune des entités appelées du graphe d'appel. Ici, nous souhaitons générer des données de test pour assurer la couverture structurelle des branches de la fonction *caisse*().

Le graphe d'appel de l'agrégat nous donne les fonctions appelées : il s'agit de *lenet*(), *Apayer*() et *prixEuro*(). Nous allons détailler la construction d'une approximation de la fonction *Apayer*() en utilisant la notation introduite au chapitre 4. Puis, dans le but d'illustrer l'impact des poids sur le modèle, nous modifierons le code de cette fonction.

5.2.1 Une approximation

Dans la fonction *Apayer*() plusieurs arcs ont la valeur d'imbrication maximale, il s'agit des arcs $a_4, a_5, a_9, a_{10}, a_{14}, a_{15}, a_{19}, a_{20}$. Nous calculons une approximation pour un arc de cette liste, par exemple $App(a_9)$.

Cette approximation correspond à l'ensemble des comportements de la fonction *Apayer*() qui passe par l'instruction *10*, $rem = 1000$ (seule instruction de l'arc a_9).

Pour calculer cette approximation, on commence par produire l'ensemble des chemins vers n_5 :

$$C_1 = \{(a_1, a_2, a_4, a_6, a_7), (a_1, a_2, a_5, a_6, a_7), (a_1, a_3, a_7)\}$$

c'est à dire tous les chemins de l'entrée de *Apayer*() jusqu'au nœud de décision précédent directement a_9 dans G.

L'ensemble des arcs et des nœuds qui interviennent dans ces chemins sont :

$$A_1 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$$

$$N_1 = \{\ e, n_1, n_2, n_3, n_4\}$$

L'ensemble des arcs de G issu de N_1 est :

$$A_0 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$$

L'ensemble des arcs supprimés pour atteindre a est :

$$A_{x0} = A_0 - A_1 = \{a_8\}$$

L'ensemble des nœuds à simplifier pour atteindre a est :

$$D_0 = \{n_4\}$$

et la simplification de ces nœuds notée $Det(D_0)$ transforme n_4 en un nœud d'assertion qui repose sur la contrainte $cont(n_4) = "ttc \geq 7500"$.

On construit ensuite le nouvel ensemble de nœuds :

$$N_1'' = N_1 - D_0 \cup Det(D_0) = \{e, n_1, n_2, n_3, n_4\}$$

où n_4 est maintenant un nœud d'assertion.

On détermine ensuite le nœud de décision n_5 dont dépend directement la couverture de a. Ce nœud devient un nœud d'assertion qui repose sur la contrainte $cont(n_5) =$ $"modePaiement == 0"$.

On construit maintenant l'ensemble des chemins qui satisfont les contraintes des nœuds d'assertion à partir de C_1 :

$$C_1' = \{(a_1, a_3, a_7)\}$$

et on construit l'ensemble des arcs et des nœuds qui y sont associés :

$$A_1' = \{a_1, a_3, a_7\}$$

$$N_1' = \{e, n_1, n_4\}$$

On a supprimé de P_1 l'ensemble des nœuds de $S = N_1 - N'_1 = \{n_2, n_3\}$.

L'ensemble des arcs supprimés par les contraintes des nouveaux nœuds d'assertion est :

$$A_{x_1} = A_1 - A'_1 = \{a_2, a_4, a_5, a_6\}$$

L'ensemble des nœuds à simplifier pour respecter les contraintes des nœuds d'assertions est :

$$D_1 = \{n_1\}$$

et la simplification de ces nœuds noté $Det(D_1)$ transforme n_1 en un nœud d'assertion qui repose sur la contrainte $cont(n_1) = "ttc < 5000||ttc \geq 7500"$.

On construit ensuite le nouvel ensemble de nœuds :

$$N'''_1 = N''_1 - D_1 \cup Det(D_1) = \{e, n_1, n_2, n_3, n_4\}$$

où n_1 et n_4 sont des nœuds d'assertion.

A ce stade, l'ensemble du graphe P_1 a été traité.

Pour traiter P_2, on construit l'ensemble des chemins issu du l'extrémité de a_9 et allant jusqu'à s :

$$\begin{aligned} C_2 \quad = \quad & \{(a_{11}, a_{12}, a_{14}, a_{16}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{12}, a_{15}, a_{16}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{12}, a_{14}, a_{16}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{12}, a_{15}, a_{16}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{12}, a_{14}, a_{16}, a_{18}, a_{22}, a_{23}), \\ & (a_{11}, a_{12}, a_{15}, a_{16}, a_{18}, a_{22}, a_{23}), \\ & (a_{11}, a_{13}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{13}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_{11}, a_{13}, a_{18}, a_{22}, a_{23})\} \end{aligned}$$

L'ensemble des chemins C'_2 est construit en concaténant les chemins de C'_1 avec ceux de C_2 autour de l'arc visé a_9 :

$$\begin{aligned} C'_2 \quad = \quad & \{(a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{14}, a_{16}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{15}, a_{16}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{14}, a_{16}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{15}, a_{16}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{14}, a_{16}, a_{18}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{12}, a_{15}, a_{16}, a_{18}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{13}, a_{17}, a_{19}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{13}, a_{17}, a_{20}, a_{21}, a_{22}, a_{23}), \\ & (a_1, a_3, a_7, a_9, a_{11}, a_{13}, a_{18}, a_{22}, a_{23})\} \end{aligned}$$

De ce nouvel ensemble, on extrait ceux qui satisfont aux contraintes des nœuds d'assertion de N_1''' :

$$C_2'' = \{(a_1, a_3, a_7, a_9, a_{11}, a_{13}, a_{18}, a_{22}, a_{23})\}$$

et l'ensemble des arcs et des n oeuds qui y interviennent :

$$A_2' = \{a_1, a_3, a_7, a_9, a_{11}, a_{13}, a_{18}, a_{22}, a_{23}\}$$

$$N_2' = \{e, n_1, n_4, n_5, n_6, n_7, n_{10}, n_{13}, n_{14}, s\}$$

Pour déterminer quels nœuds doivent être simplifiés dans P_2 à cause des contraintes des nœuds d'assertion de P_1, on répète les même étapes que précédemment :

$$A_2'' = \{a_1, a_2, a_3, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{17}, a_{18}, a_{22}, a_{23}\}$$

$$A_{x_2} = A_2'' - A_2' = \{a_2, a_8, a_{10}, a_{12}, a_{17}\}$$

$$D_2 = \{n_7, n_{10}\}$$

On simplifie ensuite les nœuds de D_2 en $Det(D_2)$. Le nœud :

- n_7 devient un nœud d'assertion reposant sur la contrainte $cont(n_7) =$ "$tcc \geq 5000 || ttc \leq 2500$",
- n_{10} devient un nœud d'assertion reposant sur la contrainte $cont(n_{10}) =$ "$ttc > 2500$",

On peut construire l'ensemble des nœuds de l'approximation comme :

$$N_2'' = N_2' - D_2 \cup Det(D_2) = \{e, n_1, n_4, n_5, n_6, n_7, n_{10}, n_{13}, n_{14}, s\}$$

où n_1, n_4, n_7, n_10 sont des nœuds d'assertion.

Finalement, l'approximation $App(a_9)$ est le graphe $G' = (N_2'', A_2', e, s)$ dont le graphe de contrôle est mis en évidence sur la figure 5.6.

Cette approximation va marquer tous les arcs de A_2'. En effet, il n'y a plus de nœuds de décision dans $App(a_9)$. Une exécution de cette approximation passe par tous les arcs qui la composent.

5.2.2 Les modèles d'entités

Le modèle complet de la fonction $Apayer()$ est constitué de 8 approximations obtenues de la même manière qu'$App(a_9)$.

Ce modèle se présente sous la forme d'une hiérarchie d'approximations de telle sorte que : $M_{Apayer} = (App(a_4), App(a_5), App(a_9), App(a_{10}), App(a_{14}), App(a_{15}), App(a_{19}), App(a_{20}))$ où $poids(App(a_4)) \leq poids(App(a_5)) \leq ... \leq poids(App(a_{20}))$.

Ici, toutes les approximations ont un poids égal. Leurs graphes de flot de contrôle sont réduits au maximum. En particulier, ils ne contiennent aucun opérateur de contrôle.

Les approximations de M_{Apayer} sont des fonctions complètement séquentielles, leur poids est directement fonction des instructions qui les composent. Avec notre exemple, ces approximations sont toutes composées du même nombre d'instructions et d'un appel à la même fonction ($lenet()$).

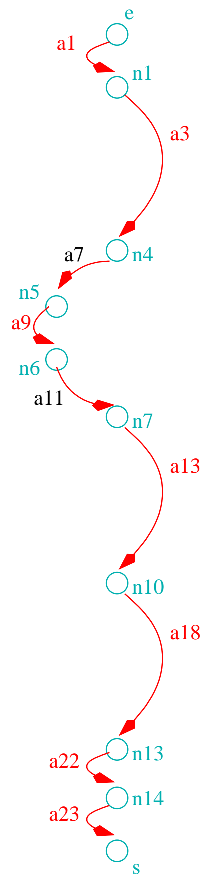

FIG. 5.6 – Mise en évidence du graphe de flot de contrôle de $App(a_9)$ sur G

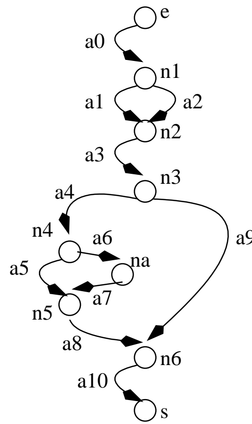

FIG. 5.7 – Graphe de flot de contrôle de la nouvelle fonction Apayer()

Pour illustrer l'impact du poids des approximations sur le modèle, imaginons que la fonction *Apayer*() soit modifiée de telle sorte :

```
short int Apayer(short int ht, short int modeP) {
        int ttc ;
        int rem ;

        if (modeP == 0) {
                rem=0 ;
        } else {
                rem=500 ;
        }

        ttc = ht - rem ;

        if (ttc > 0) {
                if (ttc >= 1000) {
                        ttc = ttc + 20 ;
                } else {
                        ttc = lenet(ttc,20) ;
                }
        } else {
                ttc = 0 ;
        }

        return ttc ;
}
```

Si on applique la méthode de construction des approximations sur cette fonction, on va extraire cinq approximations dont les graphes de contrôle sont représentés sur la figure 5.8.

Ces approximations contiennent des opérateurs de contrôle. En particulier parce que, dans cette fonction, deux des opérateurs *if...then...else* sont complétement indépendants. Le premier repose sur une décision portant sur l'entrée *modeP* tandis que le second porte

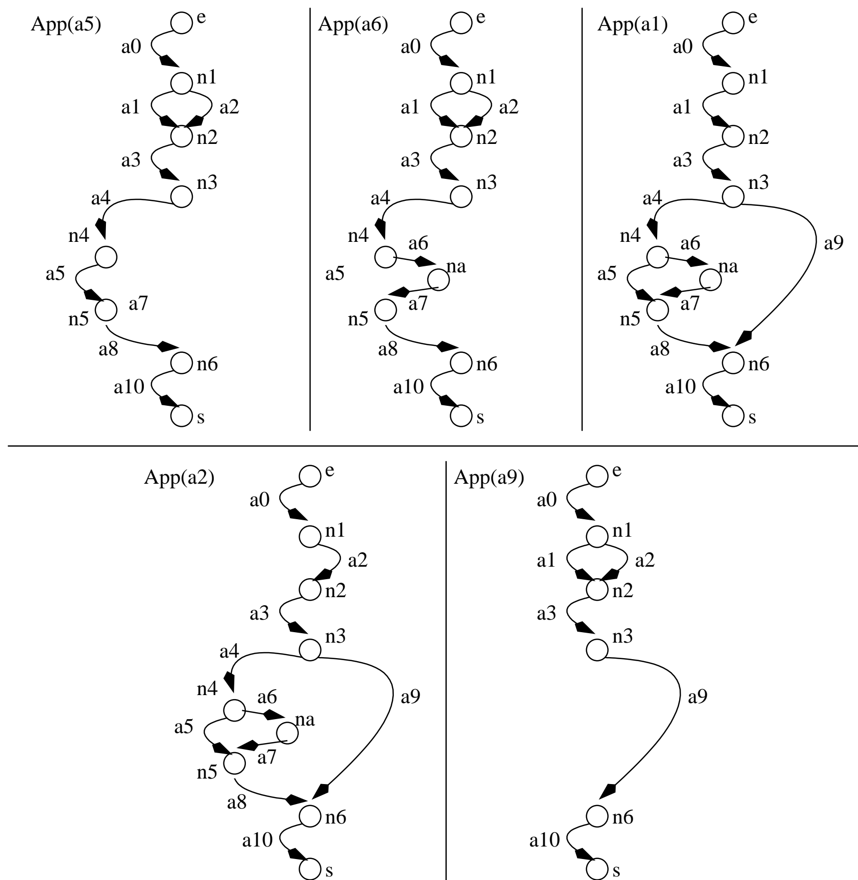

FIG. 5.8 – Les approximations issues de la fonction Apayer() modifiée

sur l'entrée *ht*.

Les poids de ces approximations seront différents. Plus précisément, le modèle de cette fonction *Apayer* modifiée est :

$$M_{Apayer()} = (App(a_9), App(a_5), App(a_6), App(a_1), App(a_2))$$

$App(a_9)$ est la plus simple des approximations, elle ne contient qu'un opérateur de contrôle dont les branches sont composées d'une seule instruction. Puis vient $App(a_5)$ qui contient une assertion de plus qu'$App(a_9)$.

Schématiquement, on peut dire que le poids de $App(a_6)$ est égal au poids de $App(a_5)$ auquel on ajoute le poids de la fonction appelée $lenet()$. Elle se trouve logiquement derrière dans la hiérarchie.

$App(a_1)$ et $App(a_2)$ ont un poids égal, le plus élevé de la hiérarchie. Elles contiennent toutes les deux un opérateur de contrôle (nœud n_3, "$ttc > 0$") dont les branches sont complexes. Elles apparaissent donc en queue du modèle.

5.3 Constructions des bouchons

Nous allons maintenant construire les bouchons de $Apayer()$ modifiée. Dans la configuration *"un bouchon par appel"*, on doit produire deux bouchons pour $Apayer()$ modifiée, chacun correspondant à un appel dans la fonction $caisse()$.

5.3.1 Calcul de l'environnement

Pour obtenir un bouchon à partir d'un modèle, il faut commencer par construire l'environnement de l'appel traité.

5.3.2 Le contexte d'appel

Les appels à $Apayer()$ se situent aux instructions *3* et *5* de la fonction $caisse()$. De plus, les entrées de $Apayer()$ sont les variables *ht* et *liquide* pour le premier appel et *ht* et *mode* pour le second.

Le contexte d'appel du premier appel à $Apayer()$ est constitué des contextes des variables *ht* et *liquide* au nœud d'appel n_3 :

Au nœud d'entrée *e* du graphe de flot de contrôle de $caisse()$ (figure 5.4), toutes les variables sont définies sur le domaine $]min, max[$ où les valeurs de *min* et *max* sont définies par le type de la variable considérée :

$$\begin{aligned}
C_{(somme,e)} &= \{]min, max[\} \\
C_{(mode,e)} &= \{]min, max[\} \\
C_{(prix,e)} &= \{]min, max[\} \\
C_{(liquide,e)} &= \{]min, max[\} \\
C_{(ht,e)} &= \{]min, max[\} \\
C_{(paiement,e)} &= \{]min, max[\}
\end{aligned}$$

Pour obtenir le contexte d'appel de $Apayer()$, les contextes de variables précédents vont évoluer à travers tous les arcs qui composent l'ensemble des chemins allant de e à n_3. Ici cet ensemble est réduit à un seul chemin :

$$C = \{(a_1, a_2, a_3)\}$$

L'évolution par l'arc a_1 nous permet d'obtenir les contextes de variables au nœud n_1. Cet arc est composé d'une seule instruction élémentaire *liquide = 1* l'ensemble des contextes résultant de cette évolution est :

$$\begin{aligned}
C_{(somme,n_1)} &= \{]min, max[\} \\
C_{(mode,n_1)} &= \{]min, max[\} \\
C_{(prix,n_1)} &= \{]min, max[\} \\
C_{(liquide,n_1)} &= \{[1, 1]\} \\
C_{(ht,n_1)} &= \{]min, max[\} \\
C_{(paiement,n_1)} &= \{]min, max[\}
\end{aligned}$$

L'arc a_2 est un arc de retour de l'appel à la fonction $prixEuro()$. Cet appel définit la variable ht. Conformément au traitement de la définition d'une variable par un appel de fonction, le domaine de ht est élargi aux bornes min et max du type de cette variable. On obtient le contexte des variables au nœud n_2 :

$$\begin{aligned}
C_{(somme,n_2)} &= \{]min, max[\} \\
C_{(mode,n_2)} &= \{]min, max[\} \\
C_{(prix,n_2)} &= \{]min, max[\} \\
C_{(liquide,n_2)} &= \{[1, 1]\} \\
C_{(ht,n_2)} &= \{]min, max[\} \\
C_{(paiement,n_2)} &= \{]min, max[\}
\end{aligned}$$

L'arc a_3 est un arc issu d'un nœud de décision, on calcul l'évolution des contextes à travers les instructions de cet arc et de l'expression symbolique qui en conditionne l'exécution. On obtient le contexte des variables au nœud n_3 :

$$\begin{aligned}
C_{(somme,n_3)} &= \{]min, max[\} \\
C_{(mode,n_3)} &= \{]min, max[\} \\
C_{(prix,n_3)} &= \{]min, max[\} \\
C_{(liquide,n_3)} &= \{[1, 1]\} \\
C_{(ht,n_3)} &= \{]0, 1000]\} \\
C_{(paiement,n_3)} &= \{]min, max[\}
\end{aligned}$$

Finalement, le contexte d'appel de $Apayer()$ en n_3 est :

$$Ca_{(Apayer,n_3)} = (C_{(ht,n_3)} = \{]0, 1000]\}, C_{(liquide,n_3)} = \{[1, 1]\})$$

En opérant de la même manière, on peut obtenir le contexte d'appel de $Apayer()$ au nœud n_5 en calculant l'évolution des domaines initiaux à travers les arcs présents sur les chemins allant de e à n_5 :

$$C = \{(a_1, a_2, a_4, a_6)\}$$

et obtenir le contexte d'appel :

$$Ca_{(Apayer,n_5)} = (C_{(ht,n_5)} = \{]1000, 10000]\}, C_{(mode,n_5)} = \{]min, max[\})$$

5.3.3 Les objectifs de génération

Pour terminer la construction des bouchons pour la génération de données de test structurel nécessite de produire les objectifs de génération des appels de $Apayer()$ dans $caisse()$.

Dans ce but, on construit l'ensemble des nœuds de décision dont l'évaluation dépend d'un appel à $Apayer()$. On obtient deux ensembles notés N_{d_1} et N_{d_2}, chacun correspondant à un appel particulier :

- pour l'appel qui à lieu au nœud n_3, $N_{d_1} = \{n_8\}$;
- pour celui qui a lieu au nœud n_5, $N_{d_2} = \{n_8\}$.

L'ensemble des arcs issus de ces nœuds sont :

$$\begin{aligned} A_1 &= \{a_{11}, a_{12}\} \\ A_2 &= \{a_{11}, a_{12}\} \end{aligned}$$

A chacun de ces arcs est associée une expression symbolique issue des nœuds des N_d qui conditionne leur exécution. Ici il s'agit de :

$$\begin{aligned} cont(a_{11}) &= \text{"}prix \leq 6000\text{"} \\ cont(a_{12}) &= \text{"}prix > 6000\text{"} \end{aligned}$$

Pour poursuivre, on calcule le contexte des variables aux nœuds de N_d. Comme N_{d_1} et N_{d_2} sont égaux, nous ne donnons qu'une seule liste de contextes obtenues, comme pour le contexte d'appel, par l'évolution du contexte initial de ces variables jusqu'au nœud d'intérêt n_8 :

$$\begin{aligned} C_{(somme,n_8)} &= \{]min, max[\} \\ C_{(mode,n_8)} &= \{]min, max[\} \\ C_{(prix,n_8)} &= \{]min, max[\} \\ C_{(liquide,n_8)} &= \{[1, 1]\} \\ C_{(ht,n_8)} &= \{(]0, 1000]), (]1000, 10000[), ([min, max])\} \\ C_{(paiement,n_8)} &= \{]min, max[\} \end{aligned}$$

On note que le contexte de la variable ht en n_8 est composé de trois domaines correspondant respectivement aux chemins $a_1, a_2, a_3, a_5, a_{10}$, $a_1, a_2, a_4, a_6, a_8, a_9, a_{10}$ et $a_1, a_2, a_4, a_7, a_9, a_{10}$ de e à n_8 dans $caise()$.

On calcule l'union de ces domaines :

$$\begin{aligned}
D_{(somme,n_8)} &= \{]min, max[\} \\
D_{(mode,n_8)} &= \{]min, max[\} \\
D_{(prix,n_8)} &= \{]min, max[\} \\
D_{(liquide,n_8)} &= \{[1, 1]\} \\
D_{(ht,n_8)} &= \{[min, max]\} \\
D_{(paiement,n_8)} &= \{]min, max[\}
\end{aligned}$$

On peut maintenant effectuer les substitutions dans $cont(a_{11})$ et $cont(a_{12})$ pour obtenir les quatre systèmes d'équations-inéquations. Chacun de ces systèmes est associé à un appel de $Apayer$ pour un arc de A_1 ou A_2.

$$\begin{aligned}
cont(Apayer_1, a11) &= \text{"}Apayer(ht, liquide) \leq 6000\text{"} \\
cont(Apayer_2, a11) &= \text{"}Apayer(ht, mode) \leq 6000\text{"} \\
cont(Apayer_1, a12) &= \text{"}Apayer(ht, liquide) > 6000\text{"} \\
cont(Apayer_2, a12) &= \text{"}Apayer(ht, mode) > 6000\text{"}
\end{aligned}$$

On note qu'ici ces systèmes sont réduits à une seule inéquation car il n'y a, pour chacun d'entre eux (i.e. pour chaque couple appel de fonction, arc traité), qu'un seul chemin de e à n_8 qui définit définit la variable $prix$.

Ces systèmes d'équations-inéquations sont les objectifs de génération des appels à $Apayer()$ dans $caisse()$:

$$\begin{aligned}
Og(Apayer_1, a11) &= \text{"}Apayer(ht, liquide) \leq 6000\text{"} \\
Og(Apayer_2, a11) &= \text{"}Apayer(ht, mode) \leq 6000\text{"} \\
Og(Apayer_1, a12) &= \text{"}Apayer(ht, liquide) > 6000\text{"} \\
Og(Apayer_2, a12) &= \text{"}Apayer(ht, mode) > 6000\text{"}
\end{aligned}$$

Pour construire le bouchon spécifique à un appel de $Apayer()$ on utilisera la liste de tous ses objetifs de génération dans $caisse()$:

$$\begin{aligned}
Og(Apayer_1, caisse) &= (Og(Apayer_1, a11), Og(Apayer_1, a12)) \\
Og(Apayer_2, caisse) &= (Og(Apayer_2, a11), Og(Apayer_2, a12))
\end{aligned}$$

5.4 Construction des bouchons dans un environnement

Maintenant que les environnements des appels à $Apayer()$ sont construits, on peut obtenir les bouchons spécifiques à ces appels par filtrage et sélection des approximations

du modèle de la fonction.

A cette fin, nous allons travailler sur le modèle de la fonction $Apayer()$ modifiée dont les approximations sont représentées sur la figure 5.8.

5.4.1 Filtrage par le contexte d'appel

Pour obtenir les bouchons sans objectifs de génération, on filtre le modèle de la fonction à partir de du contexte d'appel spécifique. On retient, dans le bouchon, les approximations valides dans ce contexte.

Le contexte du premier appel à $Apayer()$ dans $caisse()$ est :

$$Ca_{(Apayer,n_3)} = (C_{(ht,n_3)} = \{]0, 1000]\}, C_{(liquide,n_3)} = \{[1, 1]\})$$

Celui du second :

$$Ca_{(Apayer,n_5)} = (C_{(ht,n_5)} = \{]1000, 10000]\}, C_{(mode,n_5)} = \{]min, max[\})$$

Le modèle de $Apayer()$ est :

$$M_{Apayer()} = (App(a_9), App(a_5), App(a_6), App(a_1), App(a_2))$$

Chacune des approximations de ce modèles a les contraintes (liées à ses nœuds d'assertion) suivantes :

- $App(a_9)$: "$ttc \leq 0$"
- $App(a_5)$: "$ttc > 0$", "$ttc \leq 1000$"
- $App(a_6)$: "$ttc > 0$", "$ttc \geq 1000$"
- $App(a_1)$: "$modeP == 0$"
- $App(a_2)$: "$ModeP! = 0$"

En propageant le contexte d'appel de $Apayer()$ dans chacune des approximations, on vérifie la validité des approximations. Pour le premier appel à $Apayer()$ on a :

- $App(a_9)$ valide : toutes les valeurs de $ht \in]0, 500]$ permettent une exécution de l'approximation.
- $App(a_5)$ valide : toutes les valeurs de $ht \in]500, 1000]$ permettent une exécution de l'approximation.
- $App(a_6)$ invalide : puisque $modeP \in [1, 1]$, $rem = 500$ et $ht - 500 \geq 1000$ n'a pas de solution dans $]0, 1000]$.
- $App(a_1)$ invalide : car $modeP \in [1, 1]$
- $App(a_2)$ valide : car $modeP \in [1, 1]$

Le bouchon sans objectifs de génération pour le premier appel à $Apayer()$ dans $caisse()$ est donc :

$$BSOg_{Apayer()_1} = (App(a_9), App(a_5), App(a_2))$$

De la même manière, pour le deuxième appel à $Apayer()$, on obtient :

- $App(a_9)$ invalide : ni $ht - 500 \leq 0$, ni $ht - 0 \leq 0$ n'admettent solutions avec $ht \in]1000, 10000]$.
- $App(a_5)$ valide : toutes les valeurs de $ht\ in]1000, 1500[$ permettent une exécution de l'approximation.

- $App(a_6)$ valide : toutes les valeurs de $ht\ in]1000, 10000[$ permettent une exécution de l'approximation.
- $App(a_1)$ valide : toutes les valeurs de $modeP\ inmin, max$ permettent une exécution de l'approximation.
- $App(a_2)$ valide : toutes les valeurs de $modeP\ inmin, max$ permettent une exécution de l'approximation.

Le bouchon sans objectifs de génération pour le deuxième appel à $Apayer()$ dans $caisse()$ est donc :

$$BSOg_{Apayer()_2} = (App(a_5), App(a_6), App(a_1), App(a_2))$$

5.4.2 Sélection pour les objectifs de génération

Pour obtenir les bouchons avec objectifs de génération de $Apayer()$ dans $caisse()$, il faut choisir, parmi les approximations de leur bouchon sans objectif de génération une approximation valide pour chaque objectif.

Il y a deux objectifs de génération pour le premier appel :

$$Og(Apayer_1, caisse) = (Og(Apayer_1, a11), Og(Apayer_1, a12))$$

avec

$$\begin{aligned} cont(Apayer_1, a11) &= \text{"}Apayer(ht, liquide) \leq 6000\text{"} \\ cont(Apayer_1, a12) &= \text{"}Apayer(ht, liquide) > 6000\text{"} \end{aligned}$$

Pour le premier objectif de génération, nous sélectionnons une approximation de $BSOg_{Apayer()_1}$ qui satisfait la contrainte $ttc \leq 6000$. $App(a_9)$ satisfait cette contrainte. Ses chemins d'exécution dans le contexte d'appel assure que sa sortie sera toujoours égale à 0. Par conséquent, elle est sélectionnée.

Pour le deuxième objectif de génération, nous sélectionnons une approximation de $BSOg_{Apayer()_2}$ qui satisfait la contrainte $ttc > 6000$. Pour les mêmes raisons que précédement $App(a_9)$ ne satisfait pas cette contrainte. Par conséquent, elle n'est pas retenue.

L'approximation $App(a_5)$ assure que ses sorties sont égales à $ht - rem + 20$ avec $ht \in]0, 1000]$. Elle ne permet donc pas de produire des sorties dans $]6000, max]$, elle n'est pas retenue. De même, la sortie de $App(a_2)$ ne peuvent pas être supérieure à 6000 dans le contexte d'appel courant.

Le bouchon avec objectifs de génération pour le premier appel à $Apayer()$ ne satisfera donc pas le deuxième objectif de génération. En réalité, le passage par cet appel à $Apayer()$ ne permet pas d'atteindre l'instruction *10* de la fonction $caisse()$.

Ce bouchon noté $BAOg_{Apayer()_1}$ est réduit à la seule approximation $App(a_9)$.

Il y a également deux objectifs de génération pour le deuxième appel :

$$Og(Apayer_2, caisse) = (Og(Apayer_2, a11), Og(Apayer_2, a12))$$

avec

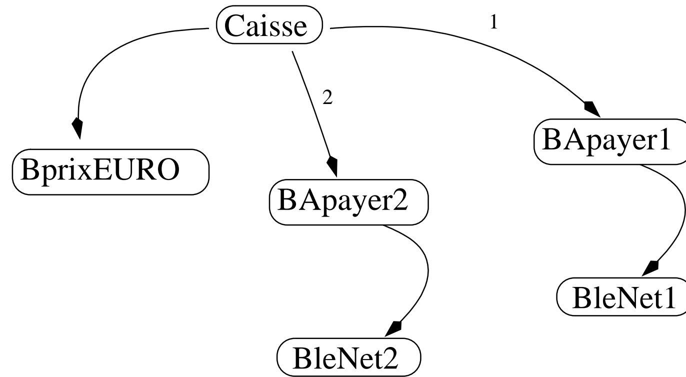

FIG. 5.9 – Les modifications effectuée sur le graphe d'appel pour utiliser les bouchons

$$\begin{aligned} Og(Apayer_2, a11) &= \text{"}Apayer(ht, mode) \leq 6000\text{"} \\ Og(Apayer_2, a12) &= \text{"}Apayer(ht, mode) > 6000\text{"} \end{aligned}$$

En effectuant les mêmes opérations que précédement, on peut dire que l'approximation $App(a_5)$ de $BSOg_{Apayer()_2}$ satisfait à la fois le premier et le deuxième objectif de génération. En effet, la donnée de test $ht = 1001, modeP = 0$ (correcte vis à vis du contexte d'appel) amène $App(a_5)$ à produire une donnée inférieure à 6000. De même, la donnée de test $ht = 10000, modeP = 0$ (correcte vis à vis du contexte d'appel) amène cette approximation à une sortie supérieure à 6000.

Comme précédement, le bouchon avec objectifs de génération du deuxième appel à $Apayer()$ noté $BAOg_{Apayer()_2}$ dans $caisse()$ est réduit à la seule approximation $App(a_5)$.

On effectue ces mêmes opérations sur les autres fonctions du graphe d'appel. On peut ensuite, après avoir modifier les appels de fonctions de sorte à les diriger vers leur bouchon respectif, lancer la génération de données de test structurel pour la fonction $caisse()$. Le graphe d'appel ainsi modifié est présenté sur la figure 5.9.

Deuxième partie

Application de la méthode sur l'outil Inka

Chapitre 6

La programmation logique avec contraintes et Inka

La programmation logique avec contraintes (PLC) [DCED96, SS94] diffère des langages usuels tel que C par sa nature déclarative. Elle a déjà été utilisée dans la création d'outil de test. Dans la majorité des cas il s'agit d'outil de test fonctionnel. En particulier, plusieurs de ces outils proposent de dériver automatiquement des cas de test à partir des spécifications de fonctions en utilisant une modélisation basée sur la programmation logique ou la programmation logique avec contraintes.

Le logiciel Inka [Got00] est un outil de génération automatique de cas de test structurel basé sur la programmation logique avec contraintes. Il utilise la puissance de déduction et la résolution de contraintes de la PLc pour effectuer le text.

Son but est de générer des données pour atteindre un composant précis de l'entité sous test. Par atteindre, on entend que si on executait le programme en lui fournissant en entrée les données générées par l'outil alors le composant spécifié dans le programme sera exécuté. Dans ce but, les entités du logiciel sont modélisées à l'aide des clauses et des contraintes de la PLC. La génération de données de test est ensuite effectuée par résolution de l'ensemble des contraintes représentant le logiciel ou un ensemble de ses entités.

La première section de ce chapitre est consacrée à une courte introduction à la programmation logique avec contraintes. Cette présentation n'a pas pour but d'être exhaustive, elle entend présenté les principaux mécanisme de ProLog qui vont nous permettre de mettre en œuvre notre implantation de la génération automatique de bouchons pour le test structurel à l'aide de l'outil Inka. La section suivante présente en détails les concepts et le fonctionnement de l'outil Inka.

6.1 Introduction à la programmation logique avec contraintes

La programmation logique avec contraintes (CLP) est un type de langage de programmation qui diffère principalement des langages classiques tel que PASCAL ou C, par sa nature déclarative. Avec un langage déclaratif, la résolution d'un problème passe par la description de sa solution et non par un algorithme. La nature déclarative de la program-

mation logique avec contraintes demande de décrire au système les données du problème (base de connaissances) à traiter sans avoir recours à des techniques algorithmiques.

La programmation logique est basée sur la logique des prédicats du premier ordre. Elle a été étendue par l'ajout de différents solveurs de contraintes. C'est ce qui a donné naissance à la programmation logique avec contraintes. Ce type de programmation combine aux processus de déduction logique des algorithmes de résolutions de contraintes essentiellement pour couper l'espace de recherche des solutions aux problèmes.

6.1.1 Quelques notions de base de ProLog

Un programme ProLog (pour PROgrammation LOGique) comporte une base de connaissances et un ensemble de règles (ou clauses).

La base de connaissance est constituée de faits qui expriment les informations connues du problème. Dans l'exemple 6.1 les faits représentent les liens de parenté entre le père, la mère et leur enfant de telle sorte que $famille(Pere, Mere, Enfant)$ soit toujours vraie.

On note que les noms françis, géraldine, cyril, aline, olivier, sylvia, jeanChristophe ainsi que famille qui commencent par une minuscule sont appelés *atomes*.

★ **Exemple 6.1** : *Une base de connaissances*

famille$(francis, geraldine, cyril)$.
famille$(aline, olivier, sylvia)$.
famille$(cyril, sylvia, jeanChristophe)$.

Les règles donnent des informations soumises à des conditions. Elles ont la forme *tête :- corps* où l'atomes : $-$ peut être lu comme *si* ou *est impliqué par*. La partie à gauche de l'atome : $-$ est appelée la tête de la régle et la partie à droite , le corps de la régle. Lorsque ProLog sait que le corps est une conséquence de la base de connaissance, il en déduit la tête de la clause.

On peut noter qu'un fait n'est qu'une régle particulière. Tous les faits d'une base de connaissance peuvent se réécrire sous la forme d'une régle où la tête de la régle est le fait et où sont corps est constitué de l'atome *true*. Ainsi le fait $famille(aline, olivier, sylvia)$ peut se réécrire $famille(aline, olivier, sylvia) : -true$.

Le corps d'une règle peut être constitué de multiples éléments ; ils peuvent être des faits ou des règles. Au sein du corps d'une règle, ils sont généralement séparés par une virgule qui exprime la *conjonction*. Toutefois il existe bien d'autres constructions entre les éléments, pour plus de détails le lecteur pourra se référer au standard prolog [DCED96].

Plusieurs règles peuvent avoir la même tête. C'est par ce mécanisme qu'on exprime la *disjonction*. Cette notation permet d'introduire des points de choix dans l'application d'une règle.

Une régle peut s'exprimer à l'aide de variables. Ces variables commencent par une majuscule. Une régle permet de définir les relations existent entre les variables apparaissant dans le corps ou la tête de la règle. On peut par exemple définir les grand parents par la règle de l'exemple 6.2.

★ **Exemple 6.2** : *Une règle*

grand_parents($PetitEnfant, GrandParents$) : $-$
famille($Pere, Mere, PetitEnfant$),
famille($GPerep, GMerep, Pere$),
famille($GPerem, GMerem, Mere$),
$GrandParents = [GPerep, GMerep, GPerem, GMerem]$,

On peut ensuite poser une question, nommé *but*, qui va provoquer une "exécution" du programme représenté par la base de connaissances et l'ensemble des règles. La recherche des solutions se fait par unification des variables et par déduction de règles.

On note que les règles apparaîssant dans le corps d'une autre règle sont appelées des sous-but dans la mesure où ils doivent être atteind pour satisfaire le but.

Par exemple, à la question

$$grand_parents(jeanChristophe, GP).$$

on obtient la réponse $GP = [francis, geraldine, aline, olivier]$.

Deux termes A et B sont unifiable par une substitution σ, si $\exists\sigma$ tel que $A\sigma = B\sigma$, par exemple $famille(A, A, A)$ et $famille(a, b, c)$ ne sont pas unifiables et $famille(A1, A2, A3)$ et $famille(A4, a, b)$ sont unifiable avec $\sigma = \{A1/A4, A2/a, A3/b\}$.

L'algorithme d'unification permet de déduire les règles applicable jusqu'à satisfaction du but. L'algorithme de résolution d'un but basé sur un ensemble de règles et une base de connaissance est décrit schématiquement par 6.1.

★ **Algorithme 6.1** : *Résolution d'un but*

Resolution :
Debut
Tant que (*le but courant n'est pas satisfait*) **faire**
Le but courant est decompose en une liste de sous buts $S_1, ..., S_i$ $i \geq 1$
Si (*une ou plusieurs regles s'appliquent a* S_1)
Alors
Choisir la premiere regle $A \leftarrow B_1, ..., B_j$ $j \geq 0$
Soit σ *tel que* $S_1 = A\sigma$
Le but devient $B_1\sigma, ..., B_j\sigma, S_2\sigma, ..., S_i\sigma$
Sinon
Retour au dernier point de choix
Finsi
FinTantque
Fin

6.1.2 Introduction des contraintes dans la programmation logique

L'ajout de contraintes à la programmation logique permet d'étendre son expressivité. Par exemple les expressions arithmétique ne peuvent s'exprimer en PL. Par exemple, le but $X+Y=Z$ en programmation logique échoue si les opérateurs $+$ et $=$ ne sont pas des régles définies dans le programme. L'unification ne pourra pas appliquer une régle et le but ne se décompose pas, la recherche de solution échoue. Résoudre ce genre de problème numérique est possible mais fastidieux en programmation logique.

Un programme logique avec contraintes est constitué d'un ensemble de régles où se mèlent des contraintes C_i et d'autres règles A_i. Ces régles ont pour forme $R : -C_1, C_2, ..., C_m \mid A_1, A_2, ..., A_n$ et le but R n'est déduit de $A_1, A_2, ..., A_n$ que si $C_1, C_2, ..., C_m$ sont satisfaites à tout instant de la décomposition des sous-buts A_i.

Les contraintes de la PLC portent sur des variables ou des listes de variables du programme. Chacune de ces variables appartient à un domaine numérique tel que les entiers ou les réels. Une contrainte est statisfaite lorsqu'elle admet une solution dans son domaine. Ces solutions, lorsqu'elles existent, sont trouvées par un solveur de contraintes spécifique au domaine des contraintes.

Il existe différents domaines de contraintes qui induisent des programmes en PLC différents. On note CLP{X} un programme de la PLC dont les contraintes admettent des solutions dans le domaine X. Le domaine particulier sur lequel le logiciel Inka cherche ses solutions s'appelle les *domaines finis*, noté CLP{FD}. Un domaine fini est un ensemble de valeurs numériques ou symboliques de cardinalité fini comme $\{1, 2, 3\}$ ou $\{a, b, c\}$.

Les contraintes de ce domaine sont linéaires $(\leq, <, =, >, \geq, \neq)$ ou symboliques qui ne sont pas liées à des opérations mathématiques comme $existe(Valeur, Liste)$ qui est vraie si $Valeur$ est présent dans la liste $Liste$.

Sur l'exemple 6.2,

$$GrandParents = [GPerep, GMerep, GPerem, GMerem]$$

est une contrainte sur les domaines finis, chacune des variables pouvant prendre des valeurs dans

$$\{francis, geraldine, cyril, aline, olivier, sylvia, jeanChristophe\}$$

L'intérêt principal de la programmation logique avec contraintes se situe dans sa manière de résoudre les problèmes. Le style déclaratif demande au programmeur de décrire les connaissances nécessaires à la résolution de son problème. Les mécanismes d'unification et de déduction produisent, à partir de la base de connaissances, l'ensemble des solutions du problème.

En contrepartie, le calcul de ces résultats peut être assez lent surtout si on le compare à une méthode classique basée sur un algorithme dans un langage impératif. L'introduction des contraintes en programmation logique permet avant tout de couper l'espace de recherche des solutions (énumération) après application des régles et part là même, de grandement accélérer le calcul des résultats.

Lorsqu'on souhaite utiliser la puissance de la programmation logique pour effectuer le test d'un logiciel, il faut nécessairement passer par une première phase de traduction des spécifications ou du code du programme en un ensemble de contraintes. On demandera ensuite au solveur de contraintes de découvrir des erreurs dans le programme en cherchant des incohérences dans l'ensemble de contraintes représentant le programme ou encore de vérifier s'il vérifie bien les propriétés qu'on a voulu imposer au programme original.

6.2 L'approche Inka : la programmation logique avec contraintes pour le test structurel

Cet outil permet d'effectuer le test unitaire de composant C d'un logiciel à partir de son code source en cherchant à le couvrir selon le critère toutes_les_branches (d'autres critères sur le graphe de flot de contrôle comme toutes_les_instructions sont possibles et des extensions de cet outil pour traiter des critères basés sur le graphe de flot de données sont étudiées dans [Got00]. De manière générale un *objectif structurel* représente une étape nécessaire dans la couverture d'un critère structurel, par conséquent sa définition exacte dépend du critère structurel choisi pour le test. Dans la suite de ce document quand aucune précision n'est donnée, le critère structurel est la couverture des branches ; un objectif structurel est donc une branche précise à atteindre dans le processus de couverture des toutes les branches.

6.2.1 Fonctionnement général

Le fonctionnement général d'INKA peut être schématisé en trois grandes étapes :

1. *la traduction* : on fournit en entrée à Inka un programme écrit en C. Ce programme contient toutes les fonctions de l'agrégat à considérer. Chacune de ces fonctions est traduite en une règle dont la tête est le nom de la fonction (précédé par $k_$). Le corps de la règle est constitué de contraintes de CLP{FD} représentant les instructions de la fonction.
2. *le filtrage* : Pour une fonction à tester donnée, on fournit à Inka un objectif structurel à atteindre. Cet objectif dépend du critère de couverture choisi pour la fonction, Inka génère des données de test pour la couverture des branches. Dans la suite de ce document, lorsqu'aucune précision n'est donnée, on parle implicitement du critère de couverture des branches.

   La juxtaposition de la programmation logique et d'un solveur de contraintes sur les domaines finis au cœur d'Inka peut alors déduire et résoudre l'ensemble des règles et des contraintes formant une fonction. Après ce filtrage, la fonction et l'objectif structurel à atteindre sont représentés par un ensemble de contraintes réduit. Il est important de noter que cet ensemble, appelé le *store* de contraintes, ne permet pas de retrouver les contraintes originales de la fonction sous une forme facilement identifiables.
3. *l'énumération* : Une fois la traduction de la fonction à tester en contraintes réduite par filtrage des contraintes, Inka va chercher à produire une donnée de test pour l'objectif structurel spécifié à l'étape précédente. En pratique l'outil va chercher une

instanciation des entrées de la fonction par énumération des domaines de chaque variable.

Pour cela, la PLC applique des heuristiques pour déterminer l'ordre d'instanciation des variables (par exemple commencer par fixer la valeur de la variable dont le domaine est le plus réduit), et la valeur à choisir dans le domaine (par exemple choisir les valeurs par ordre croissant ou décroissant).

L'énumération se termine lorsque le temps alloué à cette opération est écoulé, qu'on ait obtenu pour chaque variable une valeur conforme à l'ensemble de contraintes réduit représentant la fonction filtré ou qu'on ait épuisé toutes les combinaisons de valeurs possibles.

Le fonctionnement général d'INKA est présenté schématiquement sur la figure 6.1.

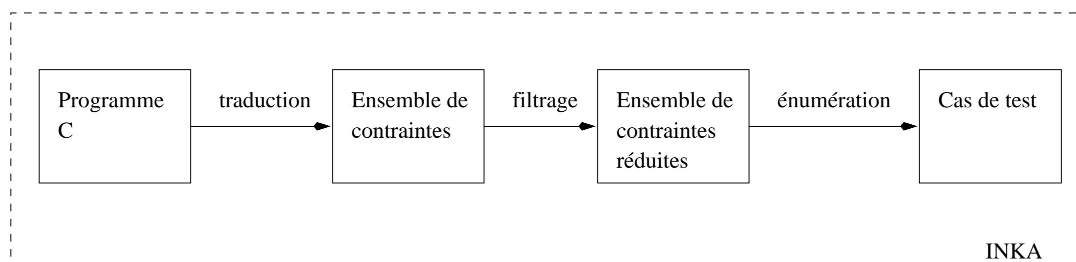

FIG. 6.1 – Fonctionnement schématique de l'outil INKA

6.2.2 Fonctionnement détaillé d'Inka

La forme SSA

La forme statique à assignation unique est une technique originalement proposée pour l'optimisation de code dans les compilateurs (voir ref. thèse arnaud p. 41) Elle transcrit le code d'une fonction en un code différent mais sémantiquement équivalent. Dans le code réécrit, chaque variable possède exactement une définition et chaque utilisation d'une variable fait référence à une définition.

A chaque nouvelle définition d'une même variable, on la renomme (v sera renommée v_0 puis v_1 etc) et chaque utilisation de la variable v fera référence au dernier renommage de la variable, ici v_1. La traduction d'une fonction dans sa forme SSA est expliquée dans la suite.

En l'abscence de pointeurs introduisant des problèmes d'aliasing, la transformation d'un bloc d'instructions sous sa forme SSA est triviale. Un problème se pose lorsque plusieurs définitions d'une même variable se rejoignent sur un nœud de jonction du graphe de flot de contrôle de la fonction ; c'est le cas pour les instructions *if then else* et *while* (l'sintruction *switch* n'est pas traité par Inka).

Pour résoudre se problème, la traduction SSA introduit une fonction particulière, nommée ϕ-fonction, qui fait correspondre le flot d'exécution de la fonction avec la bonne définition des variables utilisées après l'opérateur. Ce mécanisme est illustré sur l'exemple 6.3 où on trouve à gauche un if then else et à droite sa forme SSA. Si à l'exécution, le flot passe par la branche then alors la ϕ-fonction retourne u_1 sinon elle retourne u_2.

★ **Exemple 6.3** : *La forme SSA d'un if then else*

$x := u;$ **if** $(x < 4)$ $\quad u := 0;\ x := 10;$ **else** $\quad u := 2;$ **fi**	$x_1 := u_0;$ **if** $(x_1 < 4)$ $\quad u_1 := 0;\ x_2 := 10;$ **else** $\quad u_2 := 2;$ **fi** $u_3 := \phi(u_1, u_2);$ $x_3 := \phi(x_2, x_1);$

Dans le cas des boucles while, les ϕ-fonctions sont placées en un point particulier : avant la condition d'entrée dans la boucle et après la première instruction de boucle. Le flot exécute ces instructions lorsqu'il quitte le corps de la boucle et avant d'effectuer le test de la condition.

Intuitivement, ce comportement permet de conserver les propriétés de la forme SSA pour chaque passage dans la boucle : on renomme les variables du corps de la boucle avant chaque nouveau dépliage des instructions de la boucle. Lorsque l'exécution quitte la boucle on peut utiliser les références à la dernière exécution de la boucle, c'est à dire aux dernières valeurs des variables.

★ **Exemple 6.4** : *La forme SSA d'une boucle while*

$j := 1;$ **while** $(i > 0)$ $\quad$**do** $\quad\quad j := j * i;$ $\quad\quad i := i - 1;$ $\quad$**od**	$j_1 := 1;$ $/ * \mathit{Emplacement\ reserve} * /$ $j_3 := \phi(j_1, j_2);$ $i_2 := \phi(i_0, i_1);$ **while** $(i_2 > 0)$ $\quad$**do** $\quad\quad j_2 := j_3 * i_2;$ $\quad\quad i_1 := i_2 - 1;$ $\quad$**od**

L'avantage essentiel de la forme SSA pour la génération de données de test structurel basée sur la PLC est double. Tout d'abord elle permet de considérer les variables d'une procédure comme des variables de la programmation logique en supprimant la mise à jour destructrice de leurs valeurs ; l'affectation de la programmation impérative n'existe pas en PLC.

Ensuite elle permet la commutativité des instructions. Chaque instruction est traduite en une contrainte ; l'ordre dans lequel ces contraintes sont évaluées n'a pas d'impact sur le résultat de l'évaluation.

Lors de la phase de traduction, le logiciel INKA réécrit les procédures du programme dans leur forme SSA et c'est cette version qui est ensuite traduite sous forme de règles et de contraintes.

Génération de contraintes pour un langage if-while

L'outil Inka traduit le programme C qui doit être testé en un ensemble de contraintes sur les domaines finis (CLP{FD}). Chaque procédure est traduite en une règle. Chaque instruction des procédures est traduite en une contrainte simple ou en un opérateur spécifique du langage.

Les expressions C du programme sont traduites dans un équivalent de CLP{FD}. On associe à chaque identificateur une variable libre, c'est à dire une nouvelle variable sur laquelle il n'y a encore aucune contrainte. On traduit les constantes C par des constantes sur les domaines finis. Enfin les opérateurs arithmétiques et relationnels sont remplacés par des symboles de CLP{FD}.

Contraintes simples : elles représentent la traduction naturelle des instructions de C en contraintes. Inka peut traduire la majeure partie des instructions du langage C selon le schéma donné ci-dessous.

- *Déclaration d'une variable* : on traduit la déclaration d'une variable par une contrainte du type $X \in [min, max]$ où min et max sont les bornes supérieures et inférieures du type de la variable X. Le prototype INKA qui a été mis à disposition pour notre travail traite les differents types entier du langage C ainsi que les chaînes de caractères. Les types flottant ainsi que les structures dynamiques ne sont pas disponible dans ce prototype. L'extension du prototype pour ces données fait l'objet d'autres sous projets du projet RNTL Inka. Les problèmes spécifiques liés à ces données ne sont donc pas abordés dans cette thèse.
- *Affectation* : on traduit l'affectation de la valeur de l'expression exp à une variable X par la contrainte $X\# = \overline{exp}$ où $\overline{exp}$ est l'expression traduite dans son équivalent CLP{FD}.
- *Décisions* : on traduit une décisions $e_1\# > e_2$ par une contrainte $\overline{e_1} > \overline{e_2}$ où $\overline{e_1}$ et $\overline{e_2}$ sont les expressions traduites dans leur équivalent CLP{FD}.

Opérateurs spécifiques : Nous présentons ici les opérateurs spécifiques de l'outil Inka qui servent à traduire le flot de contrôle des fonctions. En pratique il s'agit de deux opérateurs ite et w qui traduisent respectivement la conditionnelle $if\ then\ else$ et la structure de boucle $while$. Techniquement il s'agit d'une forme particulière de contraintes, nommées *contraintes globales*, dont le fonctionnement permet à Inka de déterminer si, à un moment donné du test, le passage dans une branche est impliqué par le store. Nous reviendrons plus en détails sur ces opérateurs dans le paragraphe suivant.

- L'opérateur ite : il traduit l'opérateur if then else. Sous sa forme SSA, il s'écrit **if** *(exp)* **then** *bloc* **else** *bloc* **fi ;** $v_2 := \phi(v_0, v_1)$. Les ϕ-fonctions servent à faire le lien entre le sous-chemin emprunté à l'exécution et les variables utilisées après la conditionnelle. On peut donc scinder les ϕ-fonctions pour les intégrer aux branches de l'opérateur. On écrit alors : **if** *(exp)* **then** *bloc ;* $v_2 = v_0$ **else** *bloc ;* $v_2 = v_1$ **fi ;**. L'opérateur if then else est traduit par une contrainte globale

$$ite(\overline{exp}, \overline{bloc_then}, \overline{bloc_else})$$

– L'opérateur w : il traduit l'instruction de boucle **while do**. Sous sa forme SSA il s'écrit $v_2 = \phi(v_0, v_1)$; **while** *(exp)* **do** *bloc* **od** où v_0 est le vecteur de variables qui atteint l'instruction **while do**, v_1 le vecteur des variables définies à l'intérieur du corps de la boucle et où v_2 est le vecteur des variables utilisées dans la boucle. Il est important de noter que, contrairement à ite/3, w/5 n'est pas une simple contrainte mais un opérateur qui va générer des contraintes à l'exécution suivant le même arbre d'exécution qu'un **while do** comme explicité sur la figure 6.2. L'opérateur w/5 n'offre donc pas de garantie de terminaison.

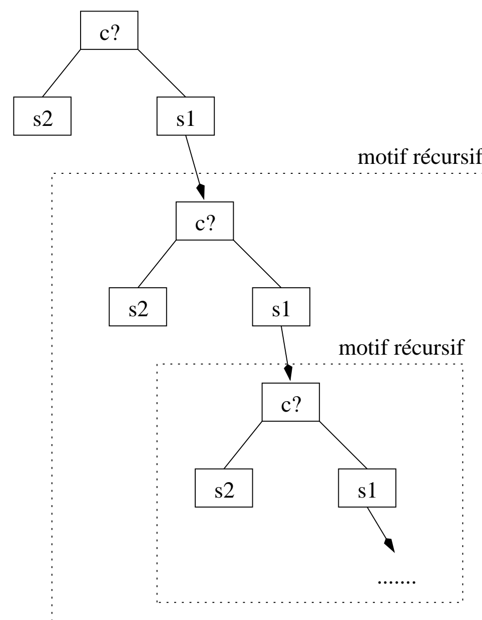

FIG. 6.2 – Arbre d'exécution d'un **while do**

Fonctionnement des opérateurs de contrôle

Dans INKA, les opérateurs de contrôle du langage source sont traduits par les opérateurs spécifiques w pour les boucles *while* et *ite* pour les opérateurs de branchement. Ces opérateurs fonctionnent par l'intermédiaire de contraintes globales.

On peut définir le fonctionnement d'une contrainte globale sous forme de condition. Lorsqu'une contrainte simple est présente dans le store, la résolution d'un but échoue si la contrainte n'est pas satisfaite à tout instant. Une contrainte globale permet de définir les conditions dans lesquelles la contrainte doit être satisfaite.

Lorsqu'une condition est vérifiée, l'évaluation de la contraintes est poursuivie sinon la contrainte est endormie et l'évaluation des conditions de vérifications de la contrainte sera réeffectuée ultérieurement.

Par exemple, dans le cas de l'opérateur *ite*, une de ses conditions pourrait être de vérifier si le store courant permet de déterminer l'exécution d'une de ses branches (then ou else). Si ça n'est pas le cas, c'est à dire qu'on ne peut pas actuellement déterminer la valeur (*vrai* ou *faux*) de la décision de l'opérateur, la contrainte globale représentant l'opérateur est endormie et cette vérification sera réeffectuée lorsque le store sera modifié.

La contrainte globale ne se réveillera que sous certaines conditions définies lors de sa création. Pour éviter d'avoir à vérifier les conditions à chaque modification du store, on définit, pour chaque contrainte globale, ses *conditions de réveil*.

Ces conditions de réveil s'expriment en terme de modification simples des valeurs des domaines des variables intervenant dans la contrainte globale. On pourra ainsi réveiller la contrainte uniquement lorsque certaines des variables qui y apparaissent seront instanciées ou, au contraire, la réveiller sur la moindre modification de leurs domaines. Par exemple, on pourrait réveiller un *ite* sur la moindre modification des domaines des variables intervenant dans sa condition.

Finalement, les contraintes globales qui sont utilisées pour traduire les opérateurs de contrôle du langage fonctionnent schématiquement en trois phases :

1. Lors de l'ajout au store de la contrainte globale, on teste si les conditions composant la contrainte sont vérifiées.
2. Si elles ne le sont pas, la contrainte globale est endormie, et ne sera réveillée que lorsque les modifications apportées au store entreront dans les conditions de réveil.
3. Si les tests réussissent, la contrainte globale est évaluée. On note que, dans ce cas, la contrainte globale n'est pas nécessairement retirée du store, elle peut être définie de telle sorte à être évaluée plusieurs fois.

Définition et appels de fonctions dans Inka

Pour faciliter la lecture de ce document nous avons fait le choix de séparer clairement le traitement des appels de fonctions du reste des étapes d'Inka.

Inka est un outil de test structurel unitaire de nature statique : il n'exécute jamais le programme sous test. Bien qu'il ne soit pas dans ses objectifs initiaux de tester des agrégats d'entités, l'outil traite les appels de fonction en ajoutant au store les contraintes de la fonction appelée lorsqu'il rencontre la tête d'une règle correspondant à une fonction appelée dans le corps de la fonction sous test. Ce fonctionnement atteint rapidement ses limites lorsque le nombre d'appels de fonctions augmente dans l'entité sous test.

Dans le logiciel Inka, on traduit un appel de fonction par une règle portant le nom de la fonction précédé de $k_$. La règle correspondant à la fonction foo à pour tête k_foo dans sa version traduite pour Inka.

– *Définition d'une fonction.* La règle associée à une fonction foo s'écrit

$$k_foo([Co],[E],[Gu],[Gd],Ret) : -C_1, C_2, ... \mid A_1, A_2, ...$$

  Sa *tête* est $k_foo([Co],[E],[Gu],[Gd],Ret)$. Les variables globales utilisées et définies par la fonction sont passées en tant que paramètres. La tête de clause d'une fonction possède cinq paramètres :

  1. Une liste de variables représentant le flot de contrôle de la fonction. Ce sont les variables de réification, nous en reparlerons en détail dans la section 6.2.3.
  2. Une liste de variables correspondant aux variables d'entrées (paramètres) de la fonction (entrées).
  3. Une liste de variables correspondant aux variables globales utilisées dans la fonction (entrées).

4. Une liste de variables correspondant aux variables globales définies dans la fonction (sorties).
5. Une variable de retour du résultat de la fonction (sortie).

– *Appel d'une fonction.* La traduction d'un appel de fonction consiste alors à écrire, dans le corps de la règle de la fonction appelante, la tête de la règle correspondant à la fonction appelée. Ses paramètres sont les paramètres traduits de la fonction appelante.
  Ainsi, si l'appel original est $V = f(3, T)$, s'il utilise une variable globale G_1 et s'il définit la variable globale résultat G_2, l'appel tranduit dans INKA sera :

$$k_f(Co, [3, T], [G_1], [G_2], RET)$$

  associé à la contrainte $V\# = RET$.

Finalement la fonction $foo()$ de l'exemple 6.5 sera traduite par le code INKA de la figure 6.6.

★ **Exemple 6.5** : *Une fonction foo*

```
unsigned short foo(unsigned short i)
{
unsigned short j;
  j = 2;
  if (i <= 16)
    j = j * i;
  if (j > 8)
    j = 0;
  return j;
}
```

★ **Exemple 6.6** : *La traduction INKA de foo*

```
k_foo([A, B, C, D], [E], [], [], F) :-
  dep(k_foo, [A, B, C, D]),
  domain([E], 0, 65535),
  domain([G], 0, 65535),
  clpfd:'t = c'(H, 2),
  ite([A, B], [E# =< 16], [E# > 16], [E, G, I, E], [J# = H * E#/\I# = J], [I# = H], K, L),
  ite([C, D], [I# > 8], [I# =< 8], [E, G, M, I], [N# = 0#/\M# = N], [M# = I], K, L),
  clpfd:'x = y'(F, M),
  true.
```

On retrouve dans le corps de la règle k_foo, la traduction des instructions C de $foo()$ en contraintes de CLP{FD}. La règle *dep*() dont la tête apparait dans le corps de k_foo est utilisée pour définir l'imbrication des opérateurs de contrôle de la fonction. Les autres éléments du corps de k_foo sont tous des contraintes de CLP{FD}.

On trouve des contraintes simples (domain, clpfd :'x=y'). On peut noter que les opérateurs arithmétiques (# =, # >, ...) et logiques (#/\, #\/, ...) portant sur les contraintes sont identifiés par #.

6.2.3 Génération d'une donnée de test

Etape de filtrage et d'énumération

On rappelle qu'on ajoute à l'ensemble des contraintes représentant une fonction, un objectif structurel à atteindre. Chaque fonction a testé est tranformée en un ensemble de contraintes la représentant. Avant de générer une donnée de test, il faut spécifier à Inka l'élément par lequel il doit passer dans la fonction. En pratique il s'agit d'une branche du GFC de la fonction. Pour cela Inka dispose d'un mécanisme lié aux contraintes globales, traduisant les opérateurs de contrôle, *ite* et w : les variables de réification.

Un vecteur de deux variables binaires $[A, B]$ est associé à chaque opérateur de contrôle de la fonction. L'objectif de ces variables est double : il permet à la fois de spécifier par quelle branche d'un opérateur de contrôle le processus de génération automatique de test doit impérativement passer et en même temps de décrire les branches par lesquelles on est effectivement passé après le filtrage et l'énumération.

Pour un *ite* : $A = 1$ signifie qu'on doit passer par la branche *then* de l'opérateur ou qu'on y est passé et $B = 1$ par sa branche else. De même que $A = 0$ signifie qu'on ne doit pas passer par la branche then ou qu'on n'y est effectivement pas passé.

Pour un w : $A = 1$ signifie qu'on doit entrer dans la boucle ou qu'on y est entré et $B = 1$ qu'on ne doit pas entrer dans la boucle ou qu'on n'y est pas entré. On peut noter que ce mécanisme ne permet pas de différencier les itérations dans une boucle. Soit le processus de test passe dans la boucle et, si le store de contrainte l'impose, les contraintes d'une boucle seront ajoutées plusieurs fois au store, soit il n'y passe pas. En l'état actuel du prototype, on ne peut pas lui spéficier le nombre d'itérations à effectuer dans une boucle.

L'ensemble des variables binaires des opérateurs de contrôle d'une fonction est reporté, dans l'ordre d'apparition dans le corps de la fonction, au niveau de sa tête : c'est le vecteur des variables de réification. Prenons l'exemple 6.5 de la fonction foo et de sa traduction dans Inka (exemple 6.6).

Dans la tête de règle $k_foo([A, B, C, D], [E], [], [], F)$, et le vecteur des variables de réification est $[A, B, C, D]$, les variables (A, B) sont associées à "**if** $(i <= 16)$" et (C, D) à "**if** $(j > 8)$". On utilise ce mécanisme pour décrire les branches à atteindre dans foo ou encore un chemin (ou un sous chemin) à parcourir dans la fonction. Ainsi, si on demande la résolution du but $A = 1, k_foo([A, B, C, D], [E], [], [], F)$, on va demander à Inka de réduire l'ensemble de contraintes représentant la fonction foo en précisant qu'on doit obligatoirement passer par la branche then de **if** $(i <= 16)$. Ce mécanisme peut être étendu à toutes les branches du GFC de la fonction foo.

Le résultat de la réduction du *store* de contraintes lors de l'étape de filtrage est un nouveau système de contraintes dans lequel le solveur CLP{FD} a réduit les domaines de toutes les variables du store original. L'étape de filtrage fournit également une valuation du vecteur des variables de réification. Ce nouveau store réduit $store'$ n'est pas disponible pour le testeur. C'est une représentation interne à ProLog des règles et des contraintes du sytème.

Pour l'utilisateur d'Inka, le résultat de l'étape de filtrage est une valuation des variables de la tête de clause de la fonction. Le résultat obtenu lorsqu'on demande la résolution du but

$$A = 1, k_foo([A, B, C, D], [E], [], [], F).$$

est :

$$\begin{array}{l} A = 1, \\ B = 0, \\ C \in 0..1, \\ D \in 0..1, \\ E \in 0..16, \\ F \in inf..sup \end{array}$$

Ceci signifie qu'Inka a déduit, à travers les solveurs de CLP{FD}, que le passage dans la branche then de **if** $(i <= 16)$ $(A = 1)$ impose :

- $B = 0$: l'impossibilité d'exécuter la branche else de **if** $(i <= 16)$.
- $C \in 0..1, D \in 0..1$: pas de contraintes sur l'exécution des branches de **if** $(j > 8)$.
- $E \in 0..16$: que la variable i de foo (E dans la traduction Inka) doit avoir une valeur comprise entre 0 et 16.
- $F \in inf..sup$: pas de contraintes sur la valeur de la variable j (F dans la traduction Inka) sortie de la fonction foo.

Pour obtenir une donnée de test, c'est à dire une valeur pour tous les paramètres de la fonction sous test, il faut ensuite trouver une valuation des variables de la fonction qui satisfasse l'ensemble des contraintes réduites de $store'$. Pour cela Inka énumère les domaines des variables de la fonction à l'aide d'heuristiques. Seules les variables de la tête de clause doivent être énumérées, les variables internes des fonctions doivent soit être initialisées, ou leurs valeurs doivent être calculés à partir des valeurs des variables de la tête de clause. Dans le cas contraire il y a une erreur dans la fonction. A chaque fois que le choix d'une valeur pour une variable est fait, on le propage à l'intérieur de $store'$ pour déterminer les conséquences de ce choix sur les autres contraintes, suivant le même algorithme que pendant le filtrage.

Dans la PLC le mécanisme de choix de valeur et celui de propagation sont entrelacés. Pour l'énumération de valeur, la PLC dispose de plusieurs heuristiques bien connues telle que : *First fail* qui consiste à choisir en premier la variable dont le domaine est le plus restreint ou encore *domain splitting* qui consiste à découper le domaine d'une variable en deux puis qui cherche à tester l'inconsistence de l'ensemble de contraintes $store'$ avec les sous domaines. Elle permet également de définir ses propres stratégies d'énumération. La statégie d'énumération d'Inka est une version circulaire du *domain splitting*.

Interprétation des résultats de l'exécution d'Inka pour générer une donnée de test

Nous avons détaillé les différentes étapes nécessaires à la génération d'un cas de test structurel à l'aide de l'outil Inka. En pratique, Inka permet de générer des cas de test assurant une couverture maximale du critère *toutes_les_branches* en sélectionnant la première branche qui n'a pas été couverte par les précédents cas de tests générés et en itérant ce processus jusqu'à avoir essayé d'exécuter ou avoir exécuté toutes les branches.

Le résultat de l'exécution d'INKA pour atteindre un point n sélectionné dans le graphe de flot de contrôle d'une fonction f peut donner trois résultats :

- L'étape de filtrage et celle d'énumération réussissent : Inka a trouvé une valuation des variables d'entrée de la fonction qui satisfait l'ensemble de contraintes réduites $store'$. Cette valuation est un cas de test qui atteint le point n de f.
- L'étape de filtrage ou d'énumération échoue. Le filtrage échoue lorsque les contraintes du $store$ sont contradictoires. L'énumération échoue lorsqu'aucune valuation satisfaisant les contraintes de $store$ n'est trouvée après avoir essayé toutes les combinaisons de valeurs possibles dans les domaines de définition (la stratégie d'énumération est complète) : dans ce cas le point n de la fonction f est non exécutable.
- L'étape de filtrage ou d'énumération ne termine pas dans le temps défini par le testeur en fixant un timer. Le processus de test est arrêté et on ne peut rien déduire de l'atteignabilité du point n de f. Ce cas de figure peut être causé par deux raisons : soit les domaines de définition des variables de la fonction sont trop larges pour être énumérés rapidement, soit l'exécution de l'opérateur w est entré dans une itération infinie. Dans le cadre général il est impossible de déterminer automatiquement laquelle de ces deux hypothèses est survenue.

6.2.4 Bénéfice et limites de l'approche

Inka associe les points forts des diverses méthodes de test structurel présentées dans la section 2.2 : c'est une méthode statique et déterministe orientée but.

Dans certains cas, lorsque le filtrage ou l'énumération conduit à un échec, cette méthode permet de prouver la non exécutabilité d'un point du programme sous test.

D'autres programmes feront échouer le processus de génération de l'outil Inka, des exemples très simples peuvent empêcher l'outil de générer une donnée de test pour un point n d'une fonction. Cette limitation est liée à la non garantie de terminaison de l'opérateur w. Des exemples de programmes simples qui amènent Inka à un échec sont présentés dans [Got00]. Il s'agit de programmes dans lesquels le filtrage ne termine pas car l'opérateur w est amené à ajouter de nouvelles contraintes au store à chaque itération.

Un autre type de programmes peut faire échouer Inka. Il s'agit de programmes dans lesquels un opérateur ite est imbriqué dans une boucle w. La génération d'une donnée de test atteignant une branche du ite imbriqué est alors fortement conditionnée par la stratégie d'énumération choisie. Sur certaines valeurs, par exemple trop grande, la déduction des nouvelles contraintes de $store'$ après le choix de telles valeurs ne terminera pas.

Dans [Got00] concernant la génération de données de test, une limite majeure du prototype identifiée est liée au traitement des appels de fonctions. Le prototype Inka

permet, lors du test unitaire d'une fonction, de prendre en considération les fonctions appelées. Comme nous l'avons vu, Inka a opté pour une représentation des fonctions à l'aide d'une clause de la PLC.

Cette représentation a l'avantage d'être transparente à la traduction : un appel de fonction est traduit par le nom de la clause représentant la fonction. Elle permet également de traiter naturellement des problèmes connus tels que les appels de fonctions récursives ou mutuellement récursives.

Lors du test unitaire d'une fonction f appelant la fonction g, l'ensemble des contraintes consituant le corps de la fonction g est posé lorsque le sous but correspondant à son appel depuis f est traité par ProLog. Ce fonctionnement est idéal du point de vue de la justesse des données de test produites. Si le point n de f dépend du résultat de l'appel à la fonction g alors, si Inka réussit à produire un cas de test atteignant n, ce cas de test atteindra n à l'exécution de la véritable application.

Pourtant ce fonctionnement dépliant tous les appels de fonctions pose rapidement un problème de complexité à Inka. On se rend facilement compte que si l'ensemble de contraintes de la fonction g comprend lui même d'autres appels de fonctions, le *store* de contraintes à analyser pour produire chaque cas de test grossit très rapidement. L'approche consistant à déplier le code de la fonction appelée à chaque appel n'est pas envisageable pour une utilisation industrielle d'Inka.

Chapitre 7

Implantation à l'aide de l'outil Inka

La méthode que nous élaborons dans cette thèse pour la création automatique de bouchons réalistes pour le test structurel basé sur le flot de contrôle ne dépend pas directement d'une implémentation particulière. La sélection d'un ensemble d'approximations suffisant pour représenter la fonction appelée peut être réalisée par n'importe quelle méthode permettant l'extraction d'un sous graphe passant par l'arc sélectionné.

Dans ce chapitre nous détaillons notre implémentation de la méthode, à l'aide de l'outil Inka décrit dans le chapitre 6.2.2.

Ainsi, nous détaillons les adaptations de la technique de création de bouchons présentée dans la partie I au contexte particulier de l'outil Inka et de la programmation logique avec contraintes (PLC). Pour cela, nous illustrons sur l'exemple du chapitre 5 les résultats obtenus avec le prototype implanté.

7.1 Choix de représentation

L'outil Inka ne travaille pas directement sur le code C des fonctions à tester. Pour tirer profit des spécificités de la PLC, cet outil commence par effectuer une traduction de C vers une représentation ProLog des fonctions à base de contraintes.

Nous avons pris le parti de nous conformer à cette approche pour notre implantation. En particulier, nous travaillons uniquement sur la traduction des fonctions effectuées par Inka.

Pour mettre en œuvre notre proposition, nous devons disposer d'un certain nombre d'informations sur l'agrégat à tester, en sus de la traduction du code des fonctions intervenant dans l'agrégat. L'agrégat doit être fourni par l'utilisateur sous la forme d'une liste de fonctions. Dans un deuxième temps, il est nécessaire de créer les différents graphes sur lesquels notre technique repose.

7.1.1 Graphe d'appel et ordre

La première étape consiste à produire, à partir du code des fonctions, le graphe d'appel de l'agrégat. Pour cela, le prototype produit une liste de faits bâtie à partir de la règle $ga(X, Y)$ où X est la tête de la règle correspondant à la fonction appelante et Y la liste des têtes de règles correspondant aux fonctions appelées par X. Cette clause est la représentation du graphe d'appel de l'agrégat.

Elle lie chaque entité aux entités qu'elle appelle. L'entité appelante $f()$ est représentée par la tête de sa clause propre k_f.

Pour construire la base de connaissance représentée par ga, on parcourt l'ensemble des "instructions" de chaque fonction et on liste les têtes des fonctions appelées. Par exemple, si la fonction $f()$ appelle les fonctions $g(), h()$, on trouvera dans la base de connaissance, le fait :

$$ga(k_f, [k_g, k_h])$$

On se base sur cet ensemble de faits pour mettre en œuvre l'algorithme de calcul d'un ordre de parcours du graphe d'appel. Nous avons implanté le calcul d'un ordre dans le cas simple où il n'y a pas de cycle dans le graphe d'appel. Cet ordre est représenté par la liste ordonnée des têtes de règles représentant les entités.

7.1.2 Graphe de flot de contrôle pondéré

Pour produire les modèles des fonctions et pour ordonner les approximations dans le modèle, nous construisons, pour chaque entité, une représentation de son graphe de flot de contrôle pondéré.

Cette représentation associe à chaque entité une liste de listes à travers la clause *poidsL*. Dans ces listes, on représente chaque bloc d'instuctions de l'entité par un poids.

Dans cette liste, on associe, à chaque opérateur de contrôle, un identifiant et, récursivement, une liste de poids pour chacun des arcs qui en est issu. On traduira ainsi un *if...then...else* par l'identificateur $if(Then, Else)$ où $Then$ et $Else$ sont les listes de listes des poids des branches de l'opérateur.

Un appel de fonction est identifié par un couple $[fct, poids]$ où fct est la tête de la clause représentant la fonction appelée et *poids* le poids de cette fonction. Le poids d'une fonction est calculé en effectuant la somme des poids de ses blocs d'instructions modifiés par leur imbrication dans un opérateur de contrôle suivant les règles présentées dans la section 4.2.3. Il est conservé par la clause *poids*.

Par exemple, pour les fonctions $lenet()$ et $APayer()$ du chapitre 5, on obtient les résultats suivants :

```
poids(k_lenet, 14).
poids(k_APayer, 76).

poidsL(k_APayer,
[11,if([if([2],[2]),1],[1]),
if([if([2],[2]),1],[1]),
if([if([2],[2]),1],[1]),
if([if([2],[2]),1],[1]),
1,[k_lenet,14],3]).

poidsL(k_lenet,
[7,if([2],[2]),1]).
```

7.1.3 Arcs et imbrication

Représentation des arcs

Pour représenter une approximation dans Inka nous avons choisi de nous appuyer sur le mécanisme des variables de réification introduit par les opérateurs de contrôle spécifiques à l'outil (voir la section 6.2.3).

Inka associe à chaque opérateur de contrôle une paire de variables $[A, B]$ permettant de spécifier quelle branche de l'opérateur doit être analysée ou quelle branche a été parcourue lors d'une analyse. Ces variables sont regroupées au niveau de la fonction dans le vecteur de réification (représenté sous forme d'une liste). Celui-ci représente tous les opérateurs de contrôle de la fonction. Inka utilise ce vecteur sous une forme bi valuée : 1 si la branche associée est analysée/exécutée, 0 si elle ne l'est pas.

L'imbrication des opérateurs de contrôle n'est pas représentée directement par cette liste. Lors de la traduction du programme C, Inka construit une clause dont les règles décrivent les dépendances entre les variables du vecteur de réification. Ces dépendances reflètent l'imbrication des opérateurs de contrôle.

Pour représenter une approximation à l'aide de ces variables, on étend le mécanisme d'INKA à une représentation tri-valuée du vecteur de réification d'une fonction. Les variables s'évaluent comme suit :

- 1 si la branche associée est présente dans l'approximation.
- 0 si la branche n'est pas présente dans l'approximation.
- _ (valeur indéfinie) lorsqu'on laisse le choix d'exécuter ou non la branche par l'approximation.

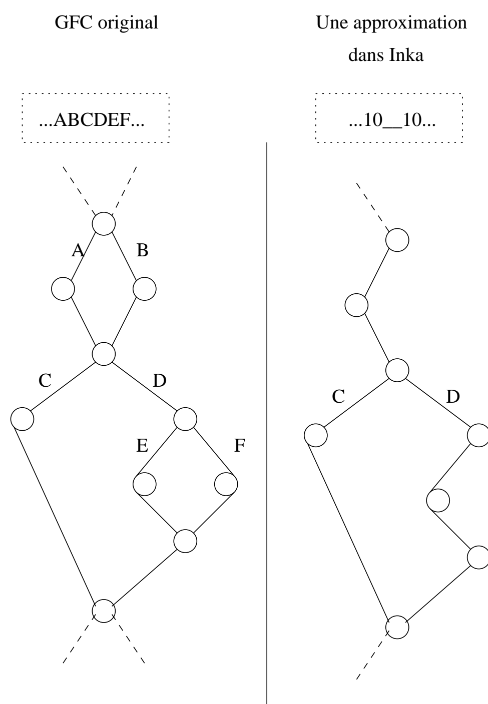

FIG. 7.1 – Une approximation représentée dans INKA

L'introduction de la valeur indéfinie permet de représenter les approximations sous

forme d'un graphe issu du graphe de flot de contrôle de la fonction. Pour représenter de manière cohérente un sous graphe à travers le vecteur de réification de la fonction, les paires de variables associées aux opérateurs de contrôle peuvent prendre n'importe quelle combinaison de valeurs dans le domaine $\{0, 1\}$ sauf dans le cas de la valeur indéfinie pour laquelle le couple doit être $[_, _]$.

La figure 7.1 illustre la représentation d'une approximation dans INKA à partir de son vecteur de réification. Elle représente la même approximation sur le graphe de flot de contrôle de la fonction.

Imbrication des arcs

L'imbrication d'un arc est donnée par l'imbrication des instructions qui le composent dans les opérateurs de contrôle du langage. A chaque opérateur de contrôle imbriqué, la valeur de l'imbrication augmente de 1 et décroît d'autant lorsqu'on arrive sur le nœud de jonction correspondant à cet opérateur.

Lorsqu'on crée une approximation dans le modèle d'une entité, on désigne l'arc dont l'approximation doit assurer la couverture. Nous avons fait le choix de désigner en priorité l'arc le plus imbriqué dans l'entité.

Donc, on construit la liste des valeurs d'imbrication des arcs de l'entité à partir de la représentation de son graphe de flot de contrôle pondéré.

Notre implantation donne la valeur d'imbrication associée à chaque variable de réification des opérateurs de contrôle. On obtient ainsi, pour la fonction $APayer()$ précédente, le vecteur d'imbrication suivant :

$$imbrication(k_APayer, [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1])$$

indiquant que les branches du premier *if...then...else* de la fonction $Apayer()$ ne sont pas imbriquées (premier couple $0, 0$). On retrouve ensuite toute la structure de l'imbrication des opérateurs de contrôle de la fonction : chaque opérateur de contrôle non imbriqué (valeur d'imbrication de 0) est immédiatement suivi d'un opérateur de contrôle imbriqué (valeur d'imbrication de 1).

On remarque que ce vecteur de valeurs d'imbrication va servir à désigner un arc unique. Il n'a donc pas besoin de refléter les dépendances d'imbrication des opérateurs de contrôle.

7.2 Modèles et bouchons

7.2.1 Construction des bouchons

Un premier prototype pour la modélisation de fonctions et la création automatique de bouchons tel que proposé dans la partie I a été implanté. Dans un premier temps, nous détaillons les apports de la PLC pour extraire les approximations d'une entité et ainsi construire son modèle.

Construction d'un modèle

Pour extraire une approximation passant par a, nous n'allons pas construire explicitement les ensembles de chemins nécessaires à la construction d'une approximation. Comme l'entité est traduite en un ensemble de contraintes, il nous suffit de désigner, par une contrainte appropriée, le fait que l'entité doit nécessairement passer par l'arc a.

Comme nous l'avons présenté, les arcs liés aux opérateurs de contrôle d'une entité sont associés à une variable accessible depuis la clause portant le nom de la fonction.

Pour forcer le passage par un arc a de l'entité, il suffit de fixer la valeur de la variable lui correspondant à 1 dans le vecteur de réification et de laisser les autres variables libres (non instanciées).

Pour extraire l'approximation passant par a, on demande la résolution du but constitué par le vecteur de réification où la variable correspondant à a est positionnée à 1 et la clause correspondant à l'entité.

Par unification et en utilisant les solveurs de contraintes Prolog, les implications du passage par a sont propagées à toute l'entité. En particulier, certaines variables non instanciées du vecteur de réification vont prendre des valeurs.

C'est ce nouveau vecteur de réification qui représente notre approximation. On calcule alors son poids en parcourant la représentation du graphe de flot de contrôle pondéré et on ajoute le couple formé par le vecteur et le poids de l'approximation qu'il représente.

Par exemple, la fonction $APayer()$ est traduite dans Inka par la clause :

$$k_APayer1([A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P], [Q], [R, S], [T], U, V, W, X)$$

où le premier paramètre est le vecteur de réification de la fonction.

Pour construire l'approximation passant par l'arc a_9 de la fonction, correspondant à la variable G du vecteur de réification, on demande la résolution du but :

$$G = 1, k_APayer1([A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P], [Q], [R, S], [T], U, V, W, X).$$

Cette résolution donne le résultat suivant :

$$
\begin{aligned}
A &= 0\\
B &= 1\\
C &= 0\\
D &= 0\\
E &= 1\\
F &= 0\\
G &= 1\\
H &= 0\\
I &= 0\\
J &= 1\\
K &= 0\\
L &= 0\\
M &= 0\\
N &= 1\\
O &= 0\\
P &= 0
\end{aligned}
$$

L'approximation $App(a_9)$ de la fonction $Apayer()$ est représentée par le vecteur de réification :

$$App(a_9) = [0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0]$$

Son poids, calculé par projection de ce vecteur sur le graphe de contrôle, est de 35. Le couple $[[0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0], 35]$ est ajouté au modèle de $Apayer()$.

Le modèle complet de la fonction est représenté par une liste de couples $(Approximation, Poids)$, pour $Apayer()$. Cela donne :

$$
\begin{aligned}
App(a_4) &= [[1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0], 35]\\
App(a_5) &= [[1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0], 35]\\
App(a_9) &= [[0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0], 35]\\
App(a_{10}) &= [[0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0], 35]\\
App(a_{14}) &= [[0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0], 35]\\
App(a_{15}) &= [[0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0], 35]\\
App(a_{19}) &= [[0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0], 35]\\
App(a_{20}) &= [[0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1], 35]
\end{aligned}
$$

On note que toutes les approximations de ce modèle ont le même poids.

Environnement

Chemins et contexte des variables Comme nous l'avons vu, le contexte d'appel est composé de la liste des domaines des entrées de la fonction appelée au nœud d'appel dans la fonction appelante.

Afin de pouvoir différencier les contextes en fonction des chemins de la fonction appelante, nous avons choisi de construire explicitement l'ensemble de ces chemins.

Pour cela, nous commençons par construire une structure équivalente à la représentation du graphe de flot de contrôle pondéré. On retient dans cette structure, la liste des instructions qui composent les arcs de l'entité. On transforme ainsi la clause de l'entité en une liste de listes d'instructions.

Cette structure a la forme suivante :

$$[Bloc, if(ListeBlocs, ListeBlocs), w(ListeBlocs), Bloc, ...]$$

où les éléments contenus dans les opérateurs de contrôle sont eux mêmes des listes de blocs. On note que les listes de blocs associées à un opérateur de contrôle contiennent chacune la contrainte conditionnant le passage par cette liste (elle se trouve en première position du premier bloc d'instructions de la liste).

Dans cette structure, les appels de fonctions sont considérés comme une simple instruction ; ils sont intégrés dans un bloc d'instructions.

Contexte d'appel Pour obtenir le contexte d'appel d'une fonction, on commence par créer la liste de tous les chemins menant à l'appel considéré en distribuant et en concaténant les blocs d'instructions menant jusqu'à l'appel de fonction.

La valeur du contexte d'appel pour chacun des chemins est obtenue par résolution du but constitué par toutes les instructions d'un chemin jusqu'à l'appel[1] et par l'utilisation des contraintes CLP{FD} $fd_max(Var, Max)$ et $fd_min(Var, Min)$ donnant respectivement la valeur Max maximale et Min minimale du domaine de la variable Var.

On obtient ainsi, pour chaque chemin, une liste de couples

$$[[V_1, [Min_1, Max_1], [V_2, [Min_2, Max_2]], ...]$$

où les V_i sont les entrées de l'entité appelée et chaque couple Min_i, Max_i correspond au domaine de cette entrée au nœud d'appel à l'entité.

Pour vérifier si l'approximation $App(a)$ de $g()$ est valide dans son contexte d'appel $Ca = [ListeCouple_{chemin_1}, ..., ListeCouple_{chemin_n}]$, on demandera la résolution successive des buts :

$$ListeCouple_{chemin_i}, App(a)$$

où les variables V_i de $ListeCouple_{chemin_i}$ seront unifiées avec les paramètres d'entrées de $App(a)$.

L'approximation sera considérée valide si au moins un des buts ne conduit pas à un échec.

Objectifs de génération La construction des objectifs de génération n'a pas été réalisé pour des raisons techniques liées à l'utilisation d'Inka.

Pour construire les objectifs de génération d'un appel de fonction, il est nécessaire de pouvoir déterminer quels nœuds de décision dépendent de l'exécution de cette fonction. Cet ensemble de nœuds N_d peut être déterminé par une analyse des dépendances entre les variables.

[1] L'appel lui même n'est pas posé pour éviter qu'il soit déplié lors de la résolution du but.

Or, la traduction des fonction faite par Inka et en particulier l'utilisation de la forme SSA rend l'analyse des dépendances fastidieuse. En effet, l'utilisation systématique de nouveaux identifiants à chaque définition d'une variable rend le suivi des variables et de leurs interactions difficiles techniquement.

Pour autant, si on construit à la main l'ensemble des nœuds de décision N_d alors, pour obtenir les objectifs de génération d'un appel de fonction, il suffit de construire la combinaison de tous les chemins depuis l'appel de la fonction jusqu'au nœud $n \in N_d$ avec chacun des arcs issus de n.

En pratique, pour un nœud n donné, on construit la liste de tous les chemins de l'appel jusqu'à n en distribuant les blocs d'instructions :

– depuis le bloc contenant l'appel de fonction (on en conservera dans ce bloc que les instructions qui suivent l'appel),
– jusqu'à l'opérateur de contrôle correspondant au nœud n.

L'objectif de génération pour n est formé par le produit cartésien de cet ensemble de buts avec l'ensemble des contraintes conditionnant l'exécution de chacun des arcs issus de n ; il s'agit de la première contrainte de chaque bloc d'instructions directement imbriqué dans l'opérateur de contrôle.

Par exemple, imaginons que la fonction appelante $f()$ permette de contruire la représentation suivante :

$$[Bloc1, if([Bloc2], [Bloc3]), Bloc4, if([Bloc5], [Bloc6]), Bloc7]$$

et qu'on cherche à construire l'objectif de génération de la fonction $g()$ appelée dans le $Bloc1$ pour le nœud de décision à l'origine de $if([Bloc5], [Bloc6])$.

On commence par couper $Bloc1$ en deux au niveau de l'appel à $g()$:

$$Bloc1 = append(BlocAvA, BlocApA)$$

où l'appel à $g()$ est la dernière instruction du bloc $BlocAvA$.

On distribue alors $BlocApA$ pour former tous les chemins menant à $if([Bloc5], [Bloc6])$:

$$\begin{array}{c} Bloc1, Bloc2, Bloc4 \\ Bloc1, Bloc3, Bloc4 \end{array}$$

L'objectif de génération est formé des quatre listes d'instructions :

$$\begin{array}{rcl} B_1 & = & append(Bloc1, Bloc2, Bloc4, [I_{(1,Bloc5)}]) \\ B_2 & = & append(Bloc1, Bloc3, Bloc4, [I_{(1,Bloc5)}]) \\ B_3 & = & append(Bloc1, Bloc2, Bloc4, [I_{(1,Bloc6)}]) \\ B_4 & = & append(Bloc1, Bloc3, Bloc4, [I_{(1,Bloc6)}]) \end{array}$$

où $I_{(i,Bloc_j)}$ est la i^{ieme} instruction du bloc $Bloc_j$.

Pour vérifier si une approximation $App(a)$ de $g()$ satisfait cet objectif de génération, on demandera la résolution successive des buts :

$$B_i, App(a)$$

où la sortie de $App(a)$ sera unifiée avec la variable utilisée dans les instructions de B_i.

L'approximation sera considérée valide si au moins un des buts ne conduit pas à un échec.

7.2.2 Autre modèle envisagé : la limitation de l'analyse en profondeur du graphe d'appel

Dans un premier temps, lors de la réalisation pratique, nous avons envisagé plusieurs solutions pour permettre la génération de données en présence d'un trop grand nombre d'appels de fonctions.

La solution consistant à considérer les fonctions appelées comme des boîtes noires uniquement exécutables est une des approches pour modéliser un grand nombre de fonctions appelées. Nous l'avons adoptée et avons défini le bouchon d'une fonction appelée sur cette base.

En pratique, nous altérons le processus de génération automatique de cas de test structurel en empêchant volontairement l'analyse des fonctions appelées à partir d'un rang donné du graphe d'appel de l'application. Pour ne pas générer de données de test fausses, on utilise ensuite l'exécution des fonctions appelées gelées pendant l'analyse comme "oracle" à la génération des données.

L'analyse du corps des fonctions (appelées et appelantes) permet de réduire l'espace de recherche des solutions pour les variables d'entrées de la fonction sous test. Lorsqu'on limite l'analyse des fonctions appelées à partir d'un niveau de profondeur pré-défini, on limite de la même manière la réduction des domaines des variables d'entrées de la fonction. Le travail de découverte d'une donnée de test satisfaisant notre critère de couverture s'en retrouve augmenté.

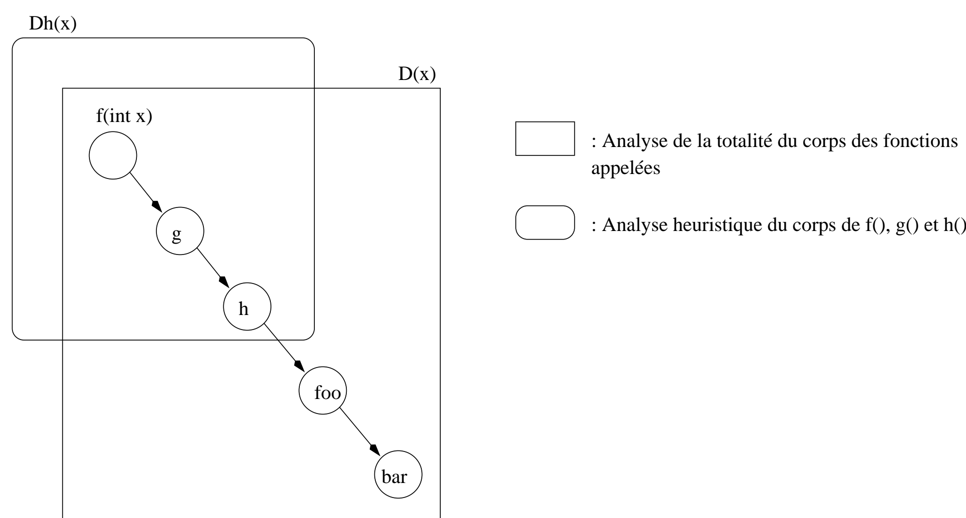

FIG. 7.2 – Limitation de l'analyse en profondeur dans le graphe d'appel

L'analyse du corps des fonctions appelées a un coût qui est loin d'être négligeable. La méthode de représentation des fonctions appelées par une boîte noire exécutable se base sur un rapport entre le temps gagné lors de l'analyse des fonctions appelées et celui perdu lors de la recherche d'une donnée de test dans l'espace de solutions désigné par l'analyse.

Ce rapport est illustré sur la figure 7.2. Cette figure représente un graphe d'appel pour le test d'une fonction $f()$ ayant un paramètre entier x. Lorsqu'on cherche à produire des données de test assurant la couverture d'un critère structurel sur $f()$, on analyse le corps des fonctions appelées g, h, foo, bar pour réduire l'espace de recherche des valeurs de x à

$D(x)$.

Effectuer la même opération en limitant l'analyse des fonctions appelées à une profondeur de 3 dans le graphe d'appel revient à analyser le corps des fonctions appelées g, h et à geler celui des fonctions foo, bar. Cette analyse réduite a une incidence sur le domaine de recherche des valeurs de x. Elle produit le domaine $Dh(x)$ tel que $D(x) \subseteq Dh(x)$.

Cette heuristique est gagnante lorsque que le temps gagné à l'analyse réduite de g, h est supérieur au temps perdu lors de la recherche d'une valeur pour x dans $Dh(x)$ au lieu de $D(x)$.

Cette méthode a été implantée au sein de l'outil Inka et fonctionne indépendamment de la création de bouchons telle qu'elle a été présentée dans la partie I. Les résultats sont encourageants bien qu'il soit difficile de déterminer a priori si cette méthode va donner de bons résultats sur un système sous test donné.

7.3 Utilisation des bouchons dans Inka

7.3.1 Opérateur de gestion des appels de fontions

Comme nous l'avons vu dans le chapitre 6.2, l'appel à une fonction f dans INKA est représenté par une règle portant le nom de cette fonction. Cette représentation a l'avantage de la simplicité. Son inconvénient principal est d'ajouter systématiquement l'ensemble des contraintes représentant f à chaque fois qu'un de ses appels est rencontré.

L'utilisation des bouchons pendant la génération de données de test structurel a été introduite dans Inka par l'utilisation d'une nouvelle contrainte globale chargée de gérer les appels de fonction.

Plus précisément, la nouvelle gestion des appels de fonctions dans Inka est confiée à deux contraintes globales. La résolution du sous but représenté par le prédicat *appel*() permet d'ajouter au store la contrainte globale définie par les paramètres donnés par le testeur :

1. *appel_solve*() est la contrainte globale qui correspond à la stratégie de limitation de l'analyse en profondeur du graphe d'appel. A partir d'une profondeur définie par le testeur à l'aide d'une variable spécifique nommée Imbrication Flag (I-Flag), la contrainte globale va geler le dépliage des fonctions appelées.

   Pour cela, la résolution de la règle *appel*() ajoute au store la contrainte *appel_solve*() avec une politique de réveil qui empêche toute analyse des fonctions appelées tant que toutes leurs entrées ne sont pas instanciées. Ceci a pour conséquence d'empêcher l'analyse des fonctions appelées[2] de profondeur supérieur au I-Flag.

2. *appel_flag*() est la contrainte globale qui applique la stratégie d'utilisation des bouchons formés d'approximation.

   La variable spécifique Model-Flag (M-Flag) du prédicat *appel*() permet au testeur de définir la taille de la fenêtre d'utilisation des bouchons réalistes.

   La contrainte globale est chargée de proposer les approximations de la fonction appelée. Lorsqu'une approximation échoue, le retour arrière de ProLog va entrainer la

[2] Les contraintes qui composent ces fonctions ne sont incluses dans le store que lorsque leurs entrées sont connues. Elles ne sont donc pas analysées mais exécutées.

suppression de l'approximation insuffisante et $appel_flag()$ fournira l'approximation suivante suivant l'ordre déterminé à la création des bouchons et jusqu'à épuisement des approximations.

Dans le cas où aucune approximation n'est disponible pour une fonction (aucun bouchon n'a été créé pour la fonction appelée ou toutes les approximations ont déjà échouées), $appel_flag()$ ajoute au store les contraintes du corps de la fonction appelée originale et se retire du store de contraintes.

Le choix du type de bouchons à utiliser lors de la génération de données de test est piloté par le prédicat *appel*(). Les drapeaux I-Flag et M-Flag ont été ajoutés à chaque tête de clause représentant une fonction de manière à répercuter le choix des bouchons en descendant dans le graphe d'appel. *appel*() détermine son comportement et la contrainte globale qu'il convient d'utiliser pour l'appel de fonction courant grâce à la valeur des flags au point d'appel.

Un appel de fontion dans la traduction originale d'Inka à la forme suivante :

$$k_fonction(C, E, GUt, GDefi, S)$$

Une fois les contraintes globales et le prédicat *appel*() implanté, il prend la forme :

$$appel(k_fonction(C, E, GUt, GDef, S, K - Flag, I - Flag, M - flag))$$

Le K-flag est un drapeau lié au mécanisme de test d'implication d'Inka. De manière à déterminer si l'ensemble des contraintes du store impose ou interdit le passage dans une branche ou une autre d'un opérateur de contrôle, Inka effectue des test d'implication en suivant des régles déterminées par la contrainte globale associée à l'opérateur.

Lors de ces tests il a été choisi de ne pas déplier les appels de fonctions pour ne pas les rendre trop long. Ce fonctionnement est implanté à travers l'instanciation du K-flag. De manière à conserver le comportement original d'Inka dans les tests d'implication, nous avons inclus la gestion du K-flag dans le prédicat *appel*().

Les autres flags vont être passés de fonction en fonction le long du graphe d'appel de l'agrégat. Le comportement du prédicat *appel*() est déterminé suivant les règles ci-dessous :

– *I-flag* : Il représente la profondeur du graphe d'appel à partir de laquelle l'opérateur appel doit geler l'appel de fonction traité en posant la contrainte globale *appel_solve*(). A chaque appel de fonction, on décrémente la valeur du I-Flag de 1 et on pose le corps de la fonction appelée.
– *M-flag* : Il représente la profondeur dans le graphe d'appel pendant laquelle l'opérateur appel utilisera les approximations de fonctions en posant la contraite globale $appel_flag()$.
  Lorsque ce paramètre est égal à 0, la contrainte $appel_flag()$ traite l'appel de fonction de la même façon que l'aurait fait Inka en posant le corps complet de la fonction appelée.
  Si la valeur du M-Flag est supérieure à 0 alors $appel_flag()$ va proposer une approximation à la place de l'appel de fonction courant. Dans le cas où l'approximation proposée contient encore des appels de fonctions alors leur M-flag sera réduit de 1 par rapport au M-flag courant de manière à modéliser la fenêtre d'utilisation des bouchons.

7.4 Expérimentation

Nous illustrons ici l'utilisation de bouchons formés d'approximations dans le cadre de la génération automatique de données de test structurel pour le critère "toutes les branches". L'exemple choisi est un pseudo additionneur binaire "64bits", deux vecteurs de 32bits chacun, où chaque bit est représenté par une variable indépendante. Cet exemple présente l'intérêt d'être simple à comprendre et de se complexifier graduellement en ajoutant des "étages" à l'additionneur. Chaque étage appelle deux fois l'étage directement inférieur.

Le génération automatique de bouchons pour toutes les fonctions du système sous test formé par les additionneur binaires de 2bits à 64bits a nécessité 69000ms. On note que dans cette illustration, les bouchons sont en réalité équivalents aux modèles. Les appels de fonctions n'ont pas d'environnement particulier qui aurait permis de simplifier les modèles.

Comparé au temps nécessaire à la couverture de toutes les fonctions sous test, la génération des bouchons représente un temps relativement court. Il est important de noter que ce temps n'est pas forcement représentatif de la complexité de la création des bouchons.

En effet, pour des raisons techniques liées à la traduction des paramètres des fonctions appelées du prototype Inka, nous avons multiplié les fonctions du système sous test. En réalité, notre agrégat est composé de 63 fonctions.

Pour chacune de ces fonctions, nous avons créé automatiquement un bouchon en appliquant la méthode présenté dans le chapitre 4, le nombre de fonctions et d'approximations pour chacune d'elles est présenté dans le tableau ci dessous :

	2bits	4bits	8bits	16bits	32bits	64bits
Nombre d'appels	32	16	8	4	2	1
Nombre d'approximations	5	4	4	4	4	4

Le tableau suivant présente les temps (en milliseconde) nécessaire à la couverture des additionneurs dans différentes configurations de la règle *appel*().

MFx / IFx	2bits		4bits		8bits		16bits		32bits		64bits	
	Fil	Enum	Fil	Enum	Fil	Enum	Fil	Enum	Fil	Enum	Fil	Enum
MF0 IF0	80	10	140	220	620	22840	2500	PTO	9670	ATO	36400	ATO
MF1 IF0	200	10	160	40	490	510	2130	22440	8940	52120	35210	ATO
MF0 IF3	80	20	160	220	650	23000	2640	708650	2360	511550	2230	537580
MF1 IF3	100	20	160	40	500	400	2120	502460	2120	505950	ATO	ATO
MF2 IF0	180	20	150	40	81280	240	98910	9460	ATO	ATO	ATO	ATO

Ici *PTO* (Partial TimeOut) signifie que la génération d'une donnée de test pour une branche particulière de la fonction a échoué en épuisant le temps qu'il lui était imparti et ATO (All TimeOut) signifie que la génération des données de test pour toutes les branches de la fonction a échoué pour la même raison. *Fil* signifie filtrage et *Enum* énumération (ce sont les deux grandes étapes de production d'une donnée de test avec l'outil Inka).

Dans ce tableau, on présente les résultats de tests de couverture effectués sur un

pseudo additionneur 64bits à chaque étage de calcul. Nous n'avons volontairement pas limité les domaines des entrées à $0, 1$ pour permettre le test de la méthode basé sur la limitation de la profondeur du graphe d'appel.

Chaque colonne donne les temps de filtrage et d'énumération pour un additionneur et les lignes présentent les différentes configurations de l'opérateur *appel*() chargé de gérer les appels de fonctions. MFx pour la fenêtre d'utilisation des bouchons avec x représentant sa taille et IFx pour la limitation dans la profondeur d'analyse du graphe d'appel avec x qui donne la profondeur au delà de laquelle les appels sont gelés.

La première ligne du tableau des temps est donnée à titre de comparaison. Aucune modification particulière n'est utilisée ; il s'agit des performances de l'outil Inka original. La deuxième ligne du tableau correspond à l'utilisation standard des bouchons réalistes avec une fenêtre de profondeur 1. Sur cet exemple on peut tirer deux constatations :

1. Tout d'abord on peut voir que la mécanique liée à l'opérateur *appel*() pénalise peu les performances de l'outil quand les approximations ne sont pas nécessaires comme c'est le cas pour add2bits. On constate dans le même temps que dès que les approximations sont utilisées, dans l'additionneur 4bits, le gain obtenu annule complètement les pertes induites par la mécanique de *appel*().
2. Ensuite on constate que l'utilisation d'une approximation à la place de la fonction originale permet de diminuer grandement le temps nécessaire à la génération automatique de donnée de test. Pour un temps de filtrage équivalent, l'utilisation du bouchon permet de diminuer le temps d'énumération nécessaire à l'obtention d'une donnée de test.

   Ceci s'explique par un filtrage plus efficace. L'approximation représente un sous ensemble des comportements possibles de la fonction originale, par conséquent elle permet d'éliminer de la fonction appelante et des fonctions appelées, les comportements inutiles dans ce contexte spécifique.

   Ceci permet à Inka utilisant les bouchons réalistes et pour un temps imparti identique de couvrir complètement l'additionneur 32bits lorsqu'Inka seul ne parvient pas à couvrir complètement l'addtionneur 16bits.

La ligne $(MF0, IF3)$ montre les temps obtenus en utilisant la technique de limitation de l'analyse en profondeur du graphe d'appel. On rappelle que cette technique a pour but de limiter le temps d'analyse des instructions des fonctions en partant du principe qu'à partir d'une profondeur fixée par le testeur, l'analyse de nouvelles instructions demande plus de temps que celui qu'on va gagner par la réduction des domaines des variables d'entrées de la fonction sous test.

Sur le tableau on peut facilement identifier l'activation de la limitation dans l'analyse, fixée arbitrairement à 3 illustrer la technique sur ce test. A partir de l'additionneur 16bits, le temps de filtrage se stabilise aux alentours de 2600ms. Dans le même temps on constate une très forte augmentation du temps d'énumération. C'est l'effet attendu de cette technique : on reporte le temps d'analyse des contraintes sur un temps d'énumération pour produire le cas de test.

Dans le cas de l'additionneur 16bits, cette heuristique n'est pas bonne. On ne gagne rien à l'analyse des instructions : le temps de filtrage est équivalent au cas de référence $(MF0, IF0)$ et à celui utilisant les bouchons réalistes $(MF1, IF0)$. Dans le même temps la durée de l'énumération est multipliée par 30 par rapport à $(MF1, IF0)$.

La technique devient intéressante dès l'analyseur 32bits, l'analyse est deux fois plus rapide que dans le cas de référence et l'énumération termine.

De même pour l'additionneur 64bits, un temps d'analyse est limité naturellement par l'heuristique et pourtant le temps d'énumération augmente peu, bien que le nombre de variables à énumérer ait été multiplié par 2. L'analyse en profondeur du graphe d'appel gèle l'analyse du corps des fonctions en dessous de l'additionneur 16bits sans augmenter le temps d'énumération.

La ligne $(MF1, IF3)$ présente le résultat de l'utilisation conjointe des deux heuristiques pour la génération de test. Les performances obtenues en combinant ces deux techniques ne sont pas bonnes.

Jusqu'à l'additionneur 8bits, la limitation de l'analyse en profondeur du graphe d'appel n'est pas active, on retrouve par conséquent des temps équivalents à ceux de la ligne $(MF1, IF0)$.

Dès l'activation de l'heuristique de limitation de l'analyse, les temps d'énumération explosent pour approcher ceux de $(MF0, IF3)$. A première vue, on pourrait croire que l'heuristique de limitation prend le dessus sur les bouchons réalistes.

Pourtant le processus de génération de données de test échoue complètement pour l'additionneur 64bits. En réalité bien que les temps obtenus pour 16 et 32bits par $(MF1, IF3)$ soient équivalents à ceux de $(MF0, IF3)$, la génération ne se passe pas de la même manière.

En observant le nombre d'échecs d'approximations proposées dans $(MF1, IF3)$, on constate que la limitation de l'analyse en profondeur du graphe d'appel empêche le processus de filtrage des contraintes de déterminer au plus tôt si une approximation est suffisante pour la génération.

De ce fait une approximation insuffisante n'est détectée qu'après avoir énuméré complètement le domaines des variables de la fonction sous test sans avoir pu trouver une donnée de test. Alors seulement, l'approximation est retirée pour proposer la suivante. Ce phénomène explique l'échec de la génération pour l'additionneur 64bits.

On peut raisonnablement en conclure que ces deux heuristiques ne doivent pas être utilisées de concert. En effet, l'heuristique basée sur l'utilisation des bouchons a besoin d'une analyse précise pour éliminer une approximation alors que la limitation de l'analyse en profondeur du graphe d'appel limite volontairement cette analyse.

La dernière ligne $(MF2, IF0)$ du tableau illustre l'impact de la fenêtre d'utilisation des bouchons. En augmentant la taille de la fenêtre d'utilisation des bouchons de 1, on constate l'effet inverse de la ligne précédente. Ici dès l'additionneur 8bits, le temps nécessaire au filtrage explose alors que le temps d'énumération est très réduit.

Ceci s'explique par le fait que le nombre d'approximations à proposer augmente de façon combinatoire avec la taille de la fenêtre. Pour l'additionneur 8bits par exemple, une fenêtre de 1 (MF1) amène dans le pire des cas à essayer 16 approximations. Lorsque cette fenêtre passe à 2, il y a 80 combinaisons d'approximations possibles.

Le temps nécessaire à trouver une combinaison suffisante pour générer la donnée de test augmente d'autant. Il est intéressant de noter que lorsqu'une approximation suffisante est proposé, l'énumération est extrêmement courte. Le système de contraintes issu du filtrage a permis de réduire au mieux les domaines des variables à énumérer.

Chapitre 8

Conclusion et perspectives

8.1 Bilan du travail

Cette thèse se place dans le cadre de la génération automatique de données de test structurel *a priori*. Pour une fonction logicielle à tester, ces données sont engendrées de manière à couvrir un composant spécifique (dans l'ensemble des composants à couvrir désignés par le critère de couverture).

Le test structurel s'opère au niveau unitaire et les fonctions appelées rencontrées pendant le test sont remplacées par des bouchons. Ces bouchons sont généralement des fonctions aux comportements très simples, pour permettre la couverture des composants sélectionnés de la fonction appelante. En contrepartie, ils ne sont généralement pas construits pour refléter les comportements de la fonction originale qu'ils remplacent.

Dans cette thèse, nous avons proposé une méthode de création automatique de bouchons pour le test structurel d'un agrégat de fonctions. Ces bouchons sont construits de manière à garantir que les valeurs founies par les bouchons sont les mêmes que celles qu'aurait produites la fonction originale dans la même situation d'appel.

Notre approche est basée sur un découpage des fonctions en approximations. Toutes les approximations d'une fonction sont regroupées et ordonnées par complexité croissante au sein d'un modèle de la fonction. Ce modèle sert de base à la construction des bouchons. La définition de l'environnement d'un appel nous permet de spécialiser le modèle de la fonction appelée pour créer le bouchon spécifique à cet appel.

L'environnement est composé d'un contexte d'appel et d'objectifs de génération. Il permet de spécialiser le modèle en filtrant les approximations qui le composent grâce au contexte d'appel puis en sélectionnant une partie de ces approximations valides. La sélection s'opère de manière à satisfaire chacun des objectifs de génération qui sont extraits de la fonction appelante.

Le modèle ainsi que les bouchons qui en sont issus sont une hiérarchie d'approximations classées par complexité d'analyse. Leur utilisation pour remplacer les fonctions appelées se fait par un parcours de la hiérarchie. L'approximation la plus simple est utilisée en premier. Si elle ne permet pas de produire une donnée de test, l'approximation suivante de la hiérarchie est proposée à sa place et ce jusqu'à épuisement des approximations du bouchons.

Finalement, notre proposition repose sur la définition d'heuristiques de poids pour hiérarchiser les approximations et sur une taille de fenêtre d'utilisation des bouchons

pour limiter les combinaisons possibles d'approximations. Les résultats pratiques que nous pouvons obtenir dépendent grandement du calibrage de ces heuristiques.

Un prototype de notre méthode de génération automatique de bouchons a été implanté à l'aide de l'outil Inka. Cet outil de génération de tests de couverture structurelle est basé sur l'utilisation de la programmation logique avec contraintes. Pour cela, l'outil traduit chaque fonction C en une règle Prolog accompagnée d'un ensemble de contraintes. Les contraintes sont ensuites résolues pour calculer les données de test.

Pour la réalisation de notre prototype, nous avons travaillé sur la base de cette traduction. La représentation des opérateurs de contrôle du langage sous forme de contraintes globales nous a permis de calculer aisement l'ensemble des approximations formant le modèle des fonctions.

Le calcul du contexte d'appel a été réalisé par construction des ensembles de contraintes, extraits de la fonction appelante, permettant de déterminer le domaine des entrées de la fonction au point d'appel. Par contre, la traduction des fonctions en un ensemble de contraintes et en particulier l'utilisation de la forme SSA se sont avérés peu pratiques pour construire automatiquement les objectifs de génération pour un appel particulier. Ils n'ont donc pas été implantés au sein de ce premier prototype.

Une première expérimentation a été effectuée sur la base de bouchons formés par les modèles des fonctions. Elle nous a permis de vérifier que cette méthode était réalisable en pratique. Les résultats obtenus sont encourageants et montrent une forte influence de la configuration du prototype sur le temps nécessaire à la génération de données.

En particulier, dans les résultats on perçoit l'impact de l'utilisation des approximations dans la fenêtre d'utilisation des bouchons. Elles tendent à allonger le temps nécessaire au filtrage des contraintes pour faire diminuer celui nécessaire à l'énumération des domaines pour trouver les données de test.

8.2 Approfondir la validation

Le prototype et l'expérimentation qui a été réalisée ne mettent pas en œuvre toutes les propositions faites dans la partie I. En particulier, les résultats d'expérimentation reflètent l'utilisation de la méthode dans le pire des cas, celui où aucun environnement ne vient simplifier les modèles de fonctions.

En pratique, nous avons déjà réalisé l'extraction d'un contexte d'appel et le filtrage d'un modèle de fonction par un contexte donné. Il reste à réaliser l'extraction automatique des objectifs de génération et la sélection des approximations les satisfaisant de manière à permettre la construction automatique complète des bouchons.

En l'état actuel du prototype Inka, le calcul des objectifs de génération est rendu compliqué par l'utilisation de la forme SSA des fonctions C. Le renommage systématique de chaque variable à chaque nouvelle définition (modification de la valeur de la variable) complique fortement la recherche des dépendances de données nécessaires à ce calcul.

Pour implanter le calcul des objectifs de génération, il faudrait envisager une meilleure intégration de notre méthode de construction des bouchons dans le fonctionnement général d'Inka. En particulier, au moment de la traduction des fonctions C en contraintes,

on pourrait effectuer un marquage des nœuds de décision dont l'évaluation dépend du résultat d'un appel de fonction. Ce marquage servirait à définir une nouvelle structure donnant pour chaque appel de la fonction $g()$ dans $f()$ la liste des nœuds de décision de $f()$ pour lesquels on doit extraire des objectifs de génération.

On pourra alors tester la méthode dans son intégralité en construisant les bouchons avec et sans objectifs de génération. On pourra alors apprécier l'efficacité de l'utilisation de l'environnement pour simplifier les modèles.

Notre méthode de génération automatique de bouchons se propose de limiter l'explosion du temps nécessaire à l'analyse du corps de fonctions appelées en les découpant pour en analyser des sous parties plus simples. La simplicité d'une approximation est estimée par une heuristique de poids associés aux instructions composant l'approximation.

Au vu des premiers résultats d'expérimentation, cette approche semble fonctionner mais on peut craindre une limitation due au nombre de combinaisons des approximations à analyser. En effet, lorsqu'on descend dans le graphe d'appel depuis la fonction sous test, tous les appels de fonctions sont replacés par un bouchon composé de multiples approximations. Lorsqu'une de ces approximations contient un appel de fonction, celui-ci est également remplacé par un bouchon. Il faut alors trouver quelle approximation "appelée" convient à l'approximation "appelante". Le nombre de combinaisons d'approximations grandit de manière combinatoire avec le nombre d'appels en cascade.

Actuellement, dans notre prototype, le nombre de ces combinaisons est limité arbitrairement par une fenêtre d'utilisation des bouchons. Ainsi, on remplacera les fonctions appelées par des bouchons sur une profondeur réduite du graphe d'appel. Ce comportement a l'intérêt de réduire le nombre de combinaisons au plus tôt dans le graphe d'appel.

Pour permettre de mieux estimer les résultats obtenus par l'application de notre méthode, il faudra dépasser les limitations actuelles du prototype Inka, en particulier en ce qui concerne les opérateurs du langage C traités.

Pendant cette thèse, en parallèle, une version pré-industrielle de l'outil Inka a été réalisée. Elle permet l'utilisation d'un spectre plus large des constructions propres du langage C, plus à même de permettre la création d'exemples. Toutefois, cette version d'Inka n'était pas à notre disposition.

8.3 Perspectives

Les perspectives de ce travail sont de deux natures : celles liées à la construction des modèles et celles liées au calcul des environnements d'appel.

Concernant le calcul de l'environnement, nous avons pris le parti, lors de l'implantation de notre méthode, de travailler entièrement sur la base de la traduction des fonctions C effectuée par le prototype Inka. De ce fait, nous nous sommes particulièrement attachés à traiter les types de données et les opérateurs déjà pris en compte par Inka.

En particulier, les données que nous avons été amenés à traiter étaient de type entier. Les autres types de données scalaires n'ont pas été abordés dans cette thèse mais leur traitement, en particulier dans le calcul des environnements ne devrait pas être différent. Ce traitement dépend essentiellement de l'utilisation de solveurs de contraintes adaptés. Par exemple, le traitement exact des flottants faisait l'objet d'un sous-projet au sein du

projet RNTL Inka.

Les structures de données classiques telles que les tableaux ou les listes posent des problèmes aux approches statiques car elles sont de nature dynamique. Dans [Got00], on propose une approche pour traduire un tableau en séparant la variable et son indice. Dans le prototype Inka, il est possible d'utiliser une variable tableau mais pas encore de la définir. De même, une traduction des relations de pointage est proposée à l'aide d'un triplet $(p, a, \{possible, definite\})$, où p pointe vers a de manière certaine ($definite$) ou possible ($possible$). Cette relation n'a pas été implantée au sein du prototype Inka.

Dans notre approche, ces structures de données posent avant tout des problèmes lors du calcul de l'environnement d'un appel. En effet, le calcul des approximations est possible dès lors que ces structures peuvent être traduites sous forme de contraintes.

Pour déterminer le domaine d'une variable appartenant à un tableau, il est nécessaire de pouvoir déterminer statiquement l'indice considéré dans le tableau. Dans le cas général, ceci n'est pas possible. Pour contourner cette difficulté au prix d'une perte de précision dans les domaines, il est toujours possible de considérer le tableau comme une seule variable dont le domaine est l'union des domaines de toutes les variables qu'il représente.

Les listes et plus généralement les structures de données dynamiques posent des problèmes qui restent ouverts. L'utilisation des pointeurs introduit des problèmes d'aliasing entre les variables ce qui rend difficile le calcul de l'évolution des domaines. Ces problèmes ouvert faisaient également l'objet d'un sous-projet de RNTL Inka. De plus, la PLC n'est pas adaptée à la définition de nouvelles structures de données qui nécessiterait l'utilisation de solveurs spécifiques à ces structures.

Pour la construction des modèles, nous nous sommes largement inspirés de la technique de slicing [Wei84]. En prenant une nouvelle définition du critère de découpe du slicing (par exemple, toutes les exécutions doivent passer par tel arc, c'est à dire tel bloc d'instructions), nous avons pu extraire d'une fonction un ensemble d'approximations. Le nombre d'approximations dépend directement du nouveau critère de découpe ; dans notre prototype il s'agit d'obtenir une couverture maximale des branches de la fonction.

Plus il y a d'approximations dans le modèle d'une fonction, plus le temps nécessaire à trouver une approximation suffisante pour la génération d'une donnée de test peut être élevé. C'est pourquoi l'utilisation de l'environnement pour simplifier les modèles est importante.

Pourtant, il peut arriver que l'environnement ne réussisse pas à éliminer certaines approximations inutiles pour réussir à générer des données de test. En particulier, pour les appels de fonctions qui ne sont pas effectués directement depuis la fonction sous test, l'environnement ne comportera pas d'objectifs de génération. C'est pourquoi nous avons introduit la notion de fenêtre d'utilisation des bouchons.

Bien que répondant au problème, cette solution est arbitraire. De plus, au sein même de la fenêtre, l'essai de combinaisons d'approximations "incompatibles" représente une part importante du temps nécessaire à générer une donnée.

Une piste pour limiter ces essais pourrait être la création d'approximations interprocédurales s'inspirant des évolutions de même type qu'a connu la technique de slicing [HRB88, SHR99]. Sans toutefois nécessiter l'utilisation du System Dependence Graph, une première expérimentation pourrait être réalisée en marquant les combinaisons d'approximations infructueuses lors de l'extraction des approximations.

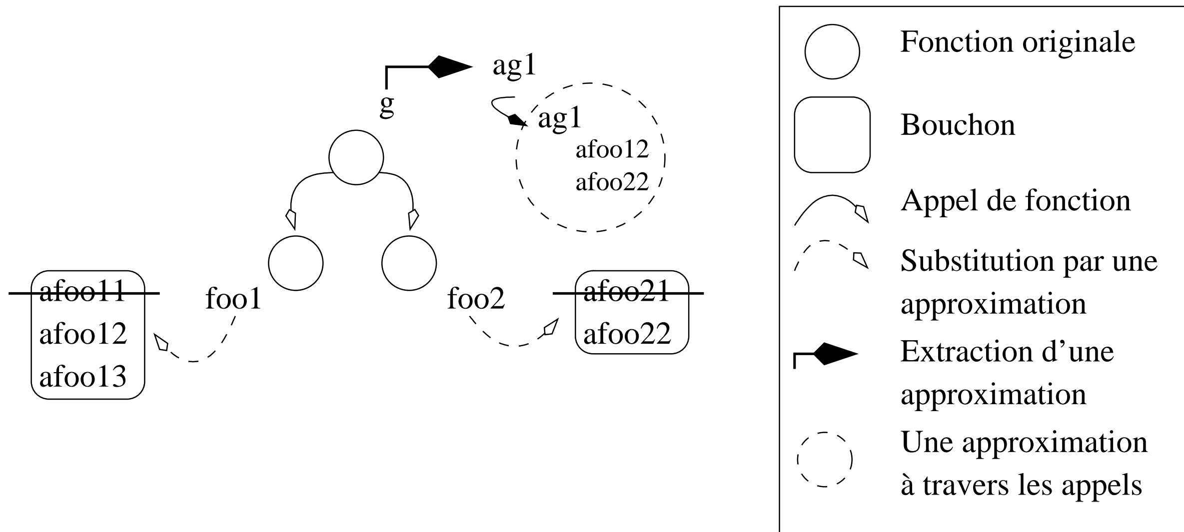

FIG. 8.1 – Une approximation inter-procédurale

En effet, la construction des bouchons est effectuée selon un ordre calculé à partir du graphe d'appel de l'agrégat. Lorsqu'on extrait l'approximation $ag1()$ de la fonction $g()$ (qui n'est pas feuille du graphe) de la figure 8.1, on peut être amené à utiliser le bouchon des fonction $foo1()$ et $foo2()$ appelées par $g()$.

On pourrait alors indiquer dans $ag1()$ qu'il n'est pas nécessaire, lors de son utilisation pour la génération de donnée, d'essayer de la combiner avec les approximations $afoo11()$ ou $afoo21()$. En effet, une telle combinaison a déjà été réfutée lors de la création de $ag1()$. Dès lors l'approximation $ag1(), afoo12(), afoo22()$ représente le graphe d'appel complet de la figure 8.1.

Bibliographie

[BLUV04] F. Bouquet, B. Legeard, M. Utting, and N. Vacelet. Faster analysis of formal specification. In J. Davies, W. Schulte, and M. Barnett, editors, *6th Int. Conf. on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 239–258, Seattle, WA, USA, November 2004. Springer-Verlag.

[CC76] Patrick Causot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, 1976.

[CC77a] Patrick Causot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 77–94, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[CD91] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, 1991.

[CP75] Cousot R. Cousot P. Vérification statique de la cohérence dynamique des programmes. Technical report, Laboratoire d'Informatique, U.S.M.G. Grenoble, 1975.

[DCED96] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog : the standard : reference manual.* Springer-Verlag, London, UK, 1996.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1) :63–86, 1996.

[GBR00] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP framework for computing structural test data. *Lecture Notes in Computer Science*, 1861 :399– ? ?, 2000.

[Got00] Arnaud Gotlieb. *Génération Automatique de Cas de Test Structurel avec la Programmation Logique Par Contraintes.* PhD thesis, Université de Nice-Sophia Antipolis, Jan 2000.

[Gri02] Karim-Cyril Griche. Automatic inter-procedural test case generation. In *Doctorial Symposium, 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, page 316, 2002.

[GWZ94] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *ISSTA '94 : Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 80–94, New York, NY, USA, 1994. ACM Press.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[JBW+94] Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. In *ISSTA '94 : Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 95–107, New York, NY, USA, 1994. ACM Press.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[KCG04] Ioannis Parissis Karim-Cyril Griche. Automatic control flow based generation of stubs for structural testing. In *IASTED International Conference on Software Engineering*, 2004.

[Kor90] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8) :870–879, 1990.

[Kor96] Bogdan Korel. Automated test data generation for programs with procedures. In *ISSTA '96 : Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215, New York, NY, USA, 1996. ACM Press.

[LCB03] Yihong Wang Lionel C. Briand, Yvan Labiche. An investigation of graph-based class integration test order strategies. *IEEE Trans. Softw. Eng.*, 29(7) :594–607, 2003.

[LPU04] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from B formal models. *Software Testing, Verification and Reliability, STVR*, 14(2) :81–103, 2004.

[Luc01] Andrea De Lucia. Program slicing : Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.

[MB89] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12) :1415–1425, 1989.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1 : Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

[P.78] Cousot P. *Méthodes itératives de construction et d'approximations de points dixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* PhD thesis, Université scientifique et médicale de Grenoble, 1978.

[RT96] Thomas Reps and Todd Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, pages 409–429, Schloss Dagstuhl, Wadern, Germany, 12–16 1996. Springer-Verlag, New York, NY.

[SHR99] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.

[SS94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.) : advanced programming techniques.* MIT Press, Cambridge, MA, USA, 1994.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3 :121–189, 1995.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4) :352–357, 1984.

Liste des tableaux

3.1 Evaluation abstraite des signes pour l'opération d'addition 38
3.2 Evaluation abstraite des signes pour l'opération de multiplication 38

4.1 Types des nœuds de f() 59
4.2 Types des arcs de f() 60

Table des figures

1.1 Un bouchon de g() 3
1.2 Un système sous test 5

2.1 Un cycle de vie en V 14
2.2 Le graphe de flot de contrôle de la procédure tordu 16
2.3 Une graphe d'appel 22

3.1 Graphe d'appel dans le cas d'un bouchon par entité appelée 31
3.2 Graphe d'appel dans le cas d'un bouchon par couple entité appelante - entité appelée 32
3.3 Graphe d'appel dans le cas d'un bouchon par appel 32
3.4 Un évaluateur partiel 36
3.5 Nœuds d'un graphe de contrôle abstrait 39

4.1 Le système sous test complet 42
4.2 Le graphe d'appel pour le test unitaire de $g2()$ 42
4.3 Processus de génération automatique de cas de test structurel 43
4.4 Graphes de flot de contrôle de f et g 44
4.5 Un exemple de graphe de flot de contrôle 48
4.6 Une approximation possible 48
4.7 Le graphe de flot de contrôle de l'approximation 49
4.8 Un graphe de flot de contrôle 50
4.9 P_1 : le sous-graphe précédent a dans G 50
4.10 P_2 : le sous-graphe suivant a dans G 51
4.11 Différentes imbrications des arcs d'un GFC 53
4.12 Architecture du modèle d'une entité 56
4.13 Deux contextes d'appels 58
4.14 Type des nœuds et des arcs de la fonction f() 59
4.15 Graphe de flot de contrôle annoté de la fonction f() 60
4.16 Quel contexte d'appel pour h ? 61
4.17 Utilisation du contexte d'appel pour les approximation d'un modèle . . . 66
4.18 Deux types de bouchons selon le rang de la fonction appelée 68
4.19 Entree du processus de génération de données de test 69
4.20 Une fenêtre d'utilisation des bouchons 70
4.21 Un graphe d'appel 71
4.22 Un ordre de calcul des bouchons 72
4.23 Un ordre possible en présence de cycles 72
4.24 Un ordre arbitraire 73

4.25 Utilisation d'un bouchon réaliste 73

5.1 Graphe de flot de contrôle de la fonction lenet() 76
5.2 Graphe de flot de contrôle de la fonction Apayer() 77
5.3 Graphe de flot de contrôle de la fonction prixEuro() 79
5.4 Graphe de flot de contrôle de la fonction caisse() 80
5.5 Le graphe d'appel de notre exemple 80
5.6 Mise en évidence du graphe de flot de contrôle de $App(a_9)$ sur G 84
5.7 Graphe de flot de contrôle de la nouvelle fonction Apayer() 85
5.8 Les approximations issues de la fonction Apayer() modifiée 86
5.9 Les modifications effectuée sur le graphe d'appel pour utiliser les bouchons 93

6.1 Fonctionnement schématique de l'outil INKA 102
6.2 Arbre d'exécution d'un **while do** 105

7.1 Une approximation représentée dans INKA 115
7.2 Limitation de l'analyse en profondeur dans le graphe d'appel 121

8.1 Une approximation inter-procédurale 131