N° d'ORDRE : 04/2013-D/inf

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'ELECTRONIQUE ET D'INFORMATIQUE

THESE

Présentée pour l'obtention du grade de doctorat
En : Informatique
Spécialité : Intelligence Artificielle et Bases de Données Avancées

Par : ALEB NASSIMA

Exécution Symbolique Arrière Pour La Vérification, Le Test Et La Génération Automatique, De Programmes

Soutenue publiquement le 21/04/2013, devant le jury composé de :

Mme Boukala Malika	**Professeur à l'USTHB**	**Présidente**
Mme Kamel Nadjet	**Maître de Conférences A à L'UFAS**	**Directrice de Thèse**
Mme Tebibel Thouraya	**Professeur à l'ESI**	**Examinateur**
Mr Boukerram Abdellah	**Professeur à L'UFAS**	**Examinateur**
Melle Boughaci Dalila	**Maître de Conférences A à L'USTHB**	**Examinateur**
Mr Boukala Cherif	**Maître de Conférences A à L'USTHB**	**Examinateur**

Sommaire

1 Introduction **6**

1.1 Contexte et motivation…………………………………………………………………….. 7

1.2 Contributions……………………………………………………………………………. 8

1.3 Organisation du document……………………………………………………………… 8

2 Background **9**

2.1 Programmes et modèles………………………………………………………. 9

2.2 Logique de Hoare et transformateurs de prédicats de Dijkstra…………........................ 10

2.2.1 Logique de Hoare……………………………………………………………………… 10

2.2.2 Transformateurs de Prédicats de Dijkstra………………………………………… 11

2.3 Exécution symbolique…………………………………………………………………....... 12

2.3.1 Définition informelle………………………………………………………………….. 12

2.3.2 Arbre d'exécution symbolique……………………………………………………….. 12

2.3.3 Limitations de l'exécution symbolique…………………………………………….. 15

2.3.4 Variantes de l'exécution symbolique…………………………………………… 15

3 Vérification de Programmes **17**

3.1 Etat de l'art…………………………………………………………………………………... 18

3.1.1 Model-checking explicite énumératif ………………………………………… 18

3.1.1.1 Problème d'explosion d'états………………………………………………. 18

3.1.1.2 Quelques solutions au problème d'explosion d'états……………………... 19

3.1.1.3 Exemples d'outils de Model-Checking explicite………………………… 19

3.1.2 Model-checking symbolique……………………………………………………….. 20

3.1.2.1 Model-checking symbolique basé sur les BDDs………………………….. 21

3.1.2.2 Model-checking symbolique borné……………………………………… 22

3.1.2.3 Model-checking basé sur l'exécution symbolique………………………… 22

3.1.3 Model-Checking basé sur l'abstraction………………………………….. 23

3.1.3.1 Interprétation abstraite …………………………………………………….. 23

3.1.3.2 Abstraction des prédicats…………………………………………… 25

3.1.4 Aspects avancés de la programmation…………………………………………… 26

3.1.4.1 Procédures……………………………………………………………………….. 26

3.1.4.2 Structures de données dynamiques……………………………………… 27

3.1.5 Model-Checking et Test ……………………………………………………………. 28

3.2 BIMC : A Backward Incremental Model-Checking………………………………………. 29

3.2.1 Principe et motivations………………………………………………………………... 31

3.2.2 Modélisation…………………………………………………………………………… 31

3.2.2.1 Pré-processing ……………………………………………………………... 32

3.2.2.2 Table des variables………………………………………………………………. 33

3.2.2.3 Table de contrôle……………………………………………………………….. 33

3.2.2.4 Propriété…………………………………………………………………………. 34

3.2.3 Description détaillée de la méthode…………………………………………………. 34

3.2.3.1 Définitions préliminaires……………………………………………………….. 38
3.2.3.2 Description de la méthode……………………………………………………… 40
3.2.3.3 Algorithme BIMC……………………………………………………………… 43
3.2.3.4 Construction de l'ensemble initial d'investigation P_0................................. 43

3.2.3.5 Analyse de Faisabilité ………………………………………………………… 45
3.2.3.6 Calcul des plus faibles préconditions………………………………………. 46
3.2.3.7 Extension de chemins ……………………………………………………….. 49
3.2.3.8 Fractionnement de chemin ………………………………………………… 49
3.2.4 Exemples illustratifs…………………………………………………………………… 50
3.2.5 Procédures et fonctions ……………………………………………………………. 55
3.2.6 Pointeurs et Alias………………………………………………………………… 57
3.2.7 Expérimentation……………………………………………………………………….. 59
3.2.8 Discussion et comparaison …………………………………………………………….. 59
3.3 Une approche évolutionnaire pour la vérification des programmes ………………… 59
3.3.1 Principe et motivations………………………………………………………… 59
3.3.2 Algorithme génétique pour la vérification de programmes………………………… 60
3.3.2.1 Algorithmes génétiques …………………………………………………….. 60
3.3.2.2 Algorithmes génétiques pour la vérification de programmes………………. 61
3.3.3 Individus……………………………………………………………………………. 63
3.3.3.1 Représentation……………………………………………………………… 63
3.3.3.2 Calcul de la chaîne d'accès d'un individu………………………………… 63
3.3.3.3 Chemin d'exécution………………………………………………………….. 64
3.3.4 Population Initiale…………………………………………………………………….. 65
3.3.5 Evaluation des individus…………………………………………………………… 66
3.3.6 Amélioration de la population………………………………………………………. 73
3.3.6.1 Croisement………………………………………………………………… 67
3.3.6.2 Mutation………………………………………………………………………... 67
3.3.7 Expérimentation………………………………………………………………… 69
3.3.8 Discussion et comparaison………………………………………………………….. 69

4 Test de Programmes **70**
4.1 Préliminaires……………………………………………………………………………… 70
4.1.1 Génération automatique de cas de test……………………………………………. 70
4.1.2 Catégories de tests………………………………………………………………… 71
4.1.2.1 Test fonctionnel……………………………………………………………… 72
4.1.2.2 Test structurel………………………………………………………………... 72
4.2 Etat de l'art : Méthodes structurelles de génération automatique de tests………………… 73
4.2.1 Génération automatique de test basée sur l'exécution symbolique……………….. 74
4.2.2 Génération automatique de test basée sur l'exécution concolique ……………….. 76
4.2.3 Approche distribuée pour la génération automatique de test……………………… 77
4.3 Nouvelle approche parallèle pour la génération de cas de test…………………………… 78
4.3.1 Exécution symbolique arrière pour la génération de cas de test…………………. 78
4.3.1.1 Modélisation des chemins d'exécution………………………………….. 78
4.3.1.2 Algorithme………………………………………………………………… 79
4.3.2 Exécution symbolique arrière parallèle pour la génération de test ……………….. 81
4.3.3 Etude de cas……………………………………………………………………… 93
4.4 Discussion………………………………………………………………………………… 84

5 Génération Automatique de Programmes **86**
5.1 Méthodologie de la programmation génétique………………………………………………. 86
5.1.1 Principe………………………………………………………………………………… 86
5.1.2 Opérateurs génétiques …………………………………………………………………. 87
5.2 Méthodes formelles pour la programmation génétique…………………………………… 89
5.3 Nouveau cadre pour la programmation génétique multi-objectifs………………………… 90
5.3.1 Principe et motivations………………………………………………………………… 90
5.3.2 Population initiale……………………………………………………………………… 90

5.3.3 Evaluation des individus………………………………………………………………. 91
5.3.3.1 Exécution symbolique des individus……………………………………….. 91
5.3.3.2 Exemples……………………………………………………………………. 92
5.3.4 Amélioration des individus……………………………………………………………. 94
5.3.4.1 Motivations …………………………………………………………………. 94
5.3.4.2 Multi-crossover……………………………………………………………… 95
5.3.4.3 Exemple d'application de l'opérateur de croisement……………………….... 97
5.4 Discussion………………………………………………………………………………. 100

6 Conclusion Générale **102**

Table des Figures

2.1 Règles de la logique de Hoare………………………………………………………………. 11

2.2 Transformateurs de Prédicats de Dijkstra………………………………………………….. 12

2.3 Exemple d'arbres d'exécution Symbolique….……………………………………………... 14

3.1 Exemple de Modélisation de Programme…………………………………………………... 34

3.2 Architecture de BIMC……………………...……………………………………………….. 38

3.3 Algorithme BIMC……………………..……………………………………………………. 39

3.4 Algorithme de calcul de l'ensemble P0 contenant la localisation erronée........................... 40

3.5 Processus d'Analyse de Faisabilité ……………………………………………………… 41

3.6 Calcul de la plus faible Précondition relativement à un élément de P… ………………… 43

3.7 Algorithme d'Analyse de Faisabilité…………………………………………………........... 44

3.8 Algorithme d'Extension d'un Chemin ………………………………………………... 45

3.9 Algorithme de Fractionnement de Chemin……………………………………………….... 46

3.10 Exemple Illustratif 2 : Programme avec Boucle et Extension de Chemin………………….. 47

3.11 Exemple Illustratif 3 : Fractionnement de Chemin ………………………………………… 49

3.12 Illustration de BIMC sur l'exemple 3………………………………………………………. 50

3.13 Sous-Programmes……………………………..……………………………………………. 52

3.14 Pointeurs………………………………………………………………………………..… 54

3.15 Résultats Expérimentaux : Backward Incremental Model-Checking…...…………………. 55

3.16 Comparaison : Programme P1…………………………………….………………………... 55

3.17 Comparaison : Programme P2…………………………………….…………………… 56

3.18 Comparaison : Programme P3 ………………...…………………………..…………….. 56

3.19 Comparaison : Programme P4 ……………………………………………………… 57

3.20 Algorithme Génétique Pour la Vérification des Programmes……………………………… 62

3.21 Opérations sur les intervalles……………………………………………………………. 65

3.22 Résultats Expérimentaux : Algorithme Génétique pour la Vérification de programmes…... 69

4.1 Algorithme BSE-Testing ……………………………... 79

4.2 Exemple Illustrant BSE ……………………………………………………………….. 80

4.3 Architecture de BSE dans un Nuage d'Ordinateurs………..………………………………. 81

4.4 Algorithme du Coordinateur…………………………….…………………………….. 82

4.5 Etude de Cas : Génération de Cas de Test dans le Cloud……..…………………………… 83

5.1 Algorithme de Croisement de deux Programmes…………………………………………. 96

5.2 Algorithme construisant VT et CT à partir d'une Formule Exclusive………….. 96

Chapitre 1
Introduction Générale

L'accroissement constant de la complexité des logiciels augmente le risque d'erreurs. La vérification des logiciels est donc indispensable pour maintenir un bon niveau de qualité et de fiabilité. Cette étape essentielle du développement des logiciels représente d'ailleurs une part importante de leur coût de production. La « Vérification de logiciel » est un terme générique regroupant : Preuve de programme, Model-Checking et Test. Les méthodes de vérification basées sur la preuve s'appuient sur la preuve au sens mathématique. Par conséquent, elles fournissent des résultats très rigoureux, toutefois, elles sont difficiles à mettre en œuvre. Le Model-Checking de logiciel a lui aussi connu un grand essor ces dernières années. Il permet de s'assurer que des propriétés sont satisfaites en des points particuliers du programme pour toutes les exécutions possibles. Quant à la dernière composante de la vérification du logiciel : Le test de logiciel, celui-ci consiste à simuler uniquement certains comportements du programme. Pour cela, le programme est exécuté avec des données d'entrée particulières. De ce fait, le test ne permet pas de garantir l'absence de fautes mais augmente la confiance portée au programme, et s'applique à toutes les phases du cycle de vie d'un logiciel. C'est la méthode de vérification la plus utilisée dans l'industrie car même s'il offre moins de garanties que la vérification basée sur la preuve et le Model-Checking, il est plus facile à mettre en œuvre. Sa pratique reste cependant trop souvent artisanale puisque dans la plupart des cas, les tests sont effectués manuellement par des ingénieurs de test. Face à ce constat, une activité de recherche très intense a vu le jour cette dernière décennie. Des modèles et des théories sont proposés pour rendre l'activité plus rigoureuse et permettre une automatisation complète du processus de test. Le test peut se faire à partir de modèle ou à partir de code. Le test basé sur le modèle vise à couvrir les éléments d'un modèle qui spécifie le comportement attendu du programme. Le test basé sur le code vise à couvrir des éléments du code du programme (instructions, branches...). Test et Model-Checking se complètent et partagent parfois des méthodes et des théories communes. L'exécution symbolique en est un exemple. La programmation génétique quant à elle s'intéresse à la génération automatique de programmes réalisant une tâche spécifique. Afin d'atteindre cet objectif, les méthodes actuelles de la programmation génétique opèrent en générant et en améliorant des

populations de programmes de façon aléatoire, jusqu'à aboutir au programme le plus 'optimal'. Par conséquent, la programmation génétique ne s'appuie sur aucun fondement formel garantissant des résultats rigoureux. L'exécution symbolique constitue un bon candidat à cette fin.

1.1.1. Contexte et Motivation

Dans cette thèse, nous nous intéressons à la vérification, la génération automatique de cas de test, et à la génération automatique de programmes. Nous utilisons une méthode d'exécution statique : l'exécution symbolique pour tous ces objectifs. Nous considérons les programmes écrits en langage C. Nous appliquons ces notions au code d'un programme et non à un modèle. Généralement, les techniques opérant directement sur le code source, représentent les programmes sous forme d'arbre. Nous avons adopté une représentation différente qui permet une manipulation aisée des programmes. Nous avons défini une approche d'exécution symbolique arrière sur cette représentation. Cette approche a été exploitée dans la vérification de propriétés de sûreté de programmes ainsi que pour la génération automatique de cas de test et la génération automatique de programmes. Ce dernier point est l'objectif de la programmation génétique. Celle-ci souffre de divers problèmes entravant son applicabilité aux problèmes réels. Nous avons défini un nouveau cadre pour la programmation génétique. L'objectif de ce cadre est de rendre plus aisée la manipulation des programmes et leur évaluation, tout en permettant de générer, à chaque nouvelle population, des programmes meilleurs que leurs prédécesseurs.

1.1.2. Contributions

Les contributions réalisées par cette thèse se résument comme suit :

1. Définition d'une nouvelle technique d'exécution symbolique: Celle-ci est orientée but, ainsi l'exploration inutile des chemins non requis à l'analyse est évitée.
2. Application de cette technique à la vérification des programmes: Ceci a donné lieu au développement de l'approche BIMC.
3. Hybridation de la méthode avec les algorithmes génétiques et l'interprétation abstraite par intervalle pour la vérification de programmes.

4. Définition et application d'une version parallèle de l'exécution symbolique arrière au est structurel de logiciels sous le critère de couverture de chemins.
5. Définition d'un nouveau cadre pour la programmation génétique multi-objectif. Celui-ci a pour but de pallier aux inconvénients des méthodes actuelles quant à la représentation, l'exécution et l'amélioration des individus programmes pour générer de façon automatiquement un programme réalisant une tâche spécifique.

1.2. Organisation du document

Par souci de rendre la lecture de ce document plus aisée, nous avons choisi de traiter les trois axes étudiés dans cette thèse : vérification, test et génération automatique de programmes, de façon indépendante. Ainsi, nous avons présenté les notions communes dans un premier chapitre dédié au background nécessaire, et nous avons consacré un chapitre par axe où figure en même temps l'état de l'art correspondant et les contributions apportées. Le reste de ce document est donc organisé comme suit : Le chapitre 2 présente quelques notions préliminaires en rapport avec cette thèse. Le chapitre 3 est consacré à la vérification de programmes : un état de l'art est présenté en premier, suivi de nos contributions dans ce domaine. Le chapitre 4 est dédié au test de programmes. Il débute par un exposé des méthodes existantes, ensuite notre méthode parallèle de génération automatique de cas de test est décrite. Le chapitre 5 est quant à lui consacré à la programmation génétique. Dans ce chapitre, nous donnons d'abord un aperçu sur la programmation génétique et ses méthodes, ainsi que les tentatives récentes liées à l'application des méthodes formelles dans ce domaine. Nous enchaînons par la description de notre méthode de génération automatique de programmes. Finalement, une conclusion générale conclut cette thèse en résumant l'ensemble des contributions et en exposant quelques perspectives qui s'y dégagent.

Chapitre 2

Background

Ce chapitre a pour objectif d'introduire quelques concepts utilisés dans la suite de cette thèse. Nous commencerons par introduire les différents formalismes utilisés pour représenter les programmes ainsi que les sémantiques qui ont été définies sur ces formalismes : la logique de Hoare [Hoa69] et les transformateurs de prédicats de Dijkstra[Dij76, Nel81]. La plus faible précondition qui est au centre de cette étude en est un exemple. Ensuite, nous exposons la technique utilisée dans cette thèse : l'exécution symbolique. Nous introduisons le principe de cette technique, ces inconvénients ainsi que les variantes qui lui ont été définies pour pallier à ses limitations.

2.1. Programmes et Modèles

L'analyse d'un programme doit généralement passer par une phase de modélisation de celui-ci. A cette fin, plusieurs formalismes ont été définis et utilisés. Ceux-ci sont généralement guidés par les concepts à utiliser pour l'analyse du programme considéré. Ainsi, les structures de kripke[Eme90] sont les modèles à utiliser quand il s'agit de vérifier des propriétés exprimées dans la logique temporelle CTL ou LTL[Eme90] . Les systèmes de transitions étiquetés sont le formalisme adéquat lorsque l'on souhaite s'exprimer en termes de sémantique opérationnelle. Les graphes de flot de contrôle (CFG) sont le formalisme le plus utilisé pour la vérification des propriétés de sûreté des programmes et la génération automatique de test.

Graphe de Flot de Contrôle [Lam83] : Un CFG : Control Flow Graph, est un quadruplet $<X,L,l_0,T>$ où : X est l'ensemble des variables du programme, L est l'ensemble des localisations de contrôle du programme, $l_0 \in L$ est la localisation initiale et T est l'ensemble des transitions : une transition t de T est de la forme (l,c,l') où l et l' sont deux localisations appartenant à L et c une contrainte sur les variables apparaissant en l et l'.

Programmes en tant que modèles

La modélisation des programmes dans l'un ou l'autre des formalismes sus cités n'est en fait pas un choix mais une nécessité imposée par le souci d'abstraction. Un bien meilleur choix est celui où le langage du programme cible et la modélisation sont exactement les mêmes. Dans ce cas, le programme peut être vérifié directement, sans la nécessité de construire un modèle séparé (le programme est lui-même utilisé comme modèle). Cela permet de réduire le coût de l'analyse en réduisant l'effort requis pour construire et maintenir des modèles distincts et explicites. Il permet également d'éviter le coût d'entretien nécessaire pour conserver les modèles et le code cohérents quand le logiciel est modifié [MSM+08]

2.2. Logique de Hoare et transformateurs de prédicats de Dijkstra

2.2.1. Logique de Hoare

La logique de Hoare [Hoa 69] est basée sur des prédicats représentant des assertions sur les états du programme.

Définition 1 (Les Triplets de Hoare). Un triplet de Hoare comprend deux prédicats P et Q, et une instruction *inst*. Le triplet de Hoare {P}*inst*{Q} signifie que si l'instruction *inst* est exécutée dans un état dans lequel P est vrai, alors Q est vrai à l'état dans lequel *inst* se termine [Nel81].
La logique de Hoare fournit les axiomes et les règles d'inférences pour raisonner sur des constructions des langages de programmation impératifs. La figure 2.1 présente quelques règles de la logique de Hoare où les notations suivantes sont utilisées :
[P] désigne la condition P, x: = e désigne l'affectation, et Assert(P) représente l'assertion P ; x est une variable du programme, e une expression sans quantificateurs et P, Q, R sont des prédicats. Un chemin π est soit la séquence vide : ε, une instruction dans le programme, ou la concaténation $\pi 1; \pi 2$ de deux chemins.

$$\frac{}{\{ P [x / e] \}\ x := e\ \{P \}}\ \textit{Affectation} \qquad \frac{}{\{ P \Rightarrow Q\}\ \ [P]\ \{Q \}}\ \textit{Condition}$$

$$\frac{}{\{ P \Rightarrow Q\}\ \ \text{assert}\ (P)\ \{Q\}}\ \textit{Assertion} \qquad \frac{\{P\}\pi_1\{Q\}\quad \{Q\}\pi_2\{R\}}{\{P\}\ \pi_1 ; \pi_2\ \ \{R \}}\ \textit{Composition}$$

$$\frac{P \Rightarrow Q \quad \{Q\}\ \pi\{R\} \quad R \Rightarrow S}{\{ P\}\pi\{S \}}\ \textit{Consequence}$$

Figure 2.1. Règles logiques de Hoare pour de simples constructions de langage de programmation.

2.2.2. Transformateurs de Prédicats de Dijkstra

Les transformateurs de prédicats sont une extension de la logique de Hoare. Dans le calcul de Dijkstra [Dij76], les états sont des fonctions totales associant des conditions préalables ou Préconditions à la Post-condition.

Définition 2 (plus forte post-condition). La plus forte post-condition SP(inst,P) pour un prédicat P par rapport à une instruction *inst* est le plus fort prédicat Q (dans le sens de l'implication) tel que {P } inst {Q}.

Définition 3 (plus faible précondition). La plus faible précondition WP(inst,Q) d'un prédicat Q relativement à l'instruction inst est le plus faible prédicat P (dans le sens de l'implication) tel que { P} inst { Q}.

Le raisonnement avec la plus forte post-condition correspond à une analyse d'accessibilité avant. Elle permet la construction des prédicats caractérisant l'ensemble des états accessibles à chaque point dans un chemin d'exécution. Tandis que le raisonnement avec la plus faible précondition correspond à une analyse arrière.

Instruction Inst	Plus forte Post-condition SP(inst, P)	Plus faible Precondition WP(inst, Q)
x:=e	$\exists x.(x=e[x/x]) \wedge P[x/x]$	Q[x/e]
[R]	$P \wedge R$	$R \Rightarrow Q$
assert(R)	$P \wedge R$	$Q \wedge R \quad (R \Rightarrow Q)$
inst1;inst2	SP(inst2, SP(inst1, P))	WP(inst1, WP(inst2, Q))

Figure 2.2. Transformateurs de Prédicats pour des instructions de programmes simples.

2.3. Exécution Symbolique

2.3.1. Définition informelle

L'exécution symbolique a été introduite par James C. King [Kin76] en tant que technique évoluée de test de programmes. Elle se réfère à la simulation de l'exécution d'un programme sur des données d'entrée représentées par des symboles. L'exécution symbolique se caractérise par le remplacement des variables d'entrée concrètes d'un programme par des symboles représentant des valeurs quelconques avec lesquelles est 'exécuté' le programme. Par conséquent, le résultat d'une exécution symbolique englobe un ensemble de cas d'exécutions possibles. Un état symbolique : état dans une exécution symbolique d'un programme, contient l'ensemble des valeurs symboliques en plus d'une condition appelée condition de chemin (Path Condition : PC) portant sur les entrées symboliques. Cette condition indique les propriétés qui doivent être satisfaites par les variables symboliques pour que l'exécution se déroule selon le chemin d'exécution associé. Une exécution symbolique débute par une condition de chemin PC=True, PC est mis à jour au long des chemins d'exécution. A la rencontre d'une nouvelle condition q, PC est transformé en $PC1=PC \wedge q$ ou $PC2=PC \wedge \neg q$ selon que la condition q doit être vraie ou fausse. Généralement les deux possibilités étant envisageables, deux chemins sont explorés en parallèle avec PC1 et PC2.

2.3.2. Arbre d'exécution symbolique

L'arbre d'exécution symbolique représente tous les chemins d'exécution suivis durant l'exécution symbolique du programme. Cet arbre est construit de la façon suivante : A chaque instruction est associé un nœud. Chaque

exécution d'une instruction conditionnelle correspond à deux transitions correspondant au cas où la condition est vraie et à celui où elle est fausse. De plus à chaque nœud est associé un état symbolique d'exécution constitué de la localisation de l'instruction dans le code du programme et de la condition de chemin PC. Les boucles sont dépliées. Ainsi, le fragment de l'arbre d'exécution symbolique associé à une itération de la boucle est dupliqué à chaque itération. Un chemin de l'arbre symbolique est faisable (feasible) si et seulement si sa formule est satisfiable. Des solveurs de contraintes sont utilisés pour résoudre les expressions symboliques représentant les conditions de chemin [DD06a, DD06b]. Un solveur de contraintes est une procédure de décision qui ayant un ensemble de contraintes trouve l'assignation adéquate des variables pour lesquelles toutes les contraintes sont simultanément satisfaites. Actuellement, il existe plusieurs types de solveurs de contraintes. Le choix d'un solveur de contrainte dépend du langage et du programme analysé. Par exemple, pour le programme de la figure 2.3, un solveur SMT : Satisfiability Modulo Theory [ES03] avec la théorie de l'arithmétique linéaire est suffisant. La performance croissante de ces outils est la principale raison du grand succès actuel de l'exécution symbolique.

Exemple 1 : Arbre d'exécution symbolique d'un programme sans boucle

Limitations de l'exécution symbolique

En dépit du fait qu'elle offre un cadre aussi formel que pratique pour l'analyse des programmes, l'exécution symbolique souffre d'obstacles entravant son application pour les programmes réels. Parmi ceux-ci :

a- Explosion du nombre de chemins d'exécution : En effet, généralement le nombre de chemins d'exécution possibles est trop grand pour être entièrement exploré. Par conséquent, l'exécution symbolique souffre de son adaptabilité aux programmes de complexité et de taille réelles.

b- L'exécution symbolique opère directement sur le code source d'un programme. Par conséquent, elle est inutilisable dans le cas d'absence du code, par exemple, dans le cas de l'utilisation des librairies.

c- Elle est assujettie aux limitations des solveurs de contraintes dont elle dépend.

d- Telle que définie initialement, l'exécution symbolique ne permet pas de manipuler tous les types de données.

2.3.3. Variantes de l'exécution symbolique

Le but initial de la définition de l'exécution symbolique par King a été le test de programmes. Celui-ci est encore le domaine privilégié de cette technique. En effet, l'arbre d'exécution symbolique est généré selon le critère de couverture souhaité du test. Pour remédier aux problèmes de l'exécution symbolique quant au développement rapide de son arbre, plusieurs variantes de celle-ci ont été définies. Celles-ci visent à résoudre notamment le problème de l'explosion des chemins et à étendre son utilisation à des programmes manipulant divers types de données. Parmi les variantes et les extensions de l'exécution symbolique :

- **Exécution symbolique généralisée**

Elle s'attaque au problème de la nature des données manipulées. Elle étend l'exécution symbolique traditionnelle pour les programmes manipulant des pointeurs. La logique de Hoare est étendue par la logique de séparation [Rey02] permettant de décrire des relations entre les prédicats d'un programme manipulant des pointeurs. Un exemple est : GSE [KPV03] et JCute [SA06b].

- **Exécution symbolique sélective**

Elle s'intéresse simultanément aux inconvénients (a) et (b) cités dans la sous section 2.3.3. Elle vise à rendre possible l'utilisation de l'exécution symbolique aux programmes utilisant des appels aux librairies. Pour analyser un programme, le code source est exécuté symboliquement, et les appels aux librairies sont exécutés concrètement. Ainsi, la taille de l'arbre d'exécution symbolique reste figée à chaque appel de la librairie. Ce genre de combinaison entre exécution concrète et symbolique est désigné par exécution concolique [MA05, SA06b]. S2e [CKC11] est un exemple marquant de ces méthodes.

- **Exécution symbolique parallèle**

Cette variante permet d'explorer les chemins d'exécution en parallèle pour permettre d'explorer le maximum de chemins d'exécution possibles. Parsym[SK10] en est un exemple.

- **Exécution symbolique arrière**

Cette variante a été utilisée pour le bytecode java. La machine virtuelle est utilisée à cet objectif. Dans [CFS09], cette technique est utilisée sur le graphe d'appels de méthodes Java.

Chapitre 3
Vérification de Programmes

Le Model-Checking est une technique algorithmique pour vérifier une description du système contre une spécification [CKL04, CGP99]. L'algorithme de Model-Checking doit soit prouver que la description du système répond à la spécification, ou bien exhiber un contre-exemple qui viole la propriété. Le software Model-Checking : Model-Checking de logiciel, se réfère à l'application des techniques du Model-Checking pour les systèmes logiciels, et en particulier sur leur mise en œuvre finale où le code source lui-même est l'objectif de l'analyse. Le code source est donc soumis à l'analyse algorithmique en vue de prouver des propriétés sur ses exécutions. Le Model-Checking de logiciel est un domaine de recherche récent et très actif [BBK+10, CCG+04, God97, Hol97]. L'entrée d'un model-checker de logiciel est le code source d'un programme et une propriété temporelle de sûreté exprimée généralement par l'instrumentation du programme [BR00, BCH+04]. Le résultat est idéalement une preuve de l'exactitude du programme ou le cas échéant, un contre-exemple, sous la forme d'un chemin d'exécution violant la propriété [HJM+02, HJM+03a]. Le Model-Checking de logiciel a été influencé par divers domaines qui ont évolué en parallèle. Le développement des logiques de programmes et des procédures de décision associées [Nel81, NO80, Sho84] a fourni un cadre et des outils de base pour raisonner sur les espaces infinis des états. Les techniques de vérification automatique de modèles [CE81, QS81, VW94] ont fourni les outils algorithmiques de base pour l'exploration de l'espace des états. L'interprétation abstraite [CC77], quant à elle, a fourni les associations entre le monde logique des espaces infinis et le monde algorithmique des représentations finies, en permettant divers niveaux d'abstraction et d'interprétation des programmes. Cependant, l'explosion de l'espace des états demeure le problème le plus crucial qui entrave sévèrement l'applicabilité des outils de Model-Checking à la vérification des logiciels. En effet, de façon théorique, le Model-Checking de programmes consiste à examiner tous les états et chemins d'exécution possibles d'une manière systématique et exhaustive afin de vérifier si une propriété est vérifiée. Cependant, dans la pratique, ceci est irréaliste car le nombre d'états augmente de façon exponentielle avec chaque variable du programme. Différentes techniques ont été définies pour remédier à cet inconvénient.

Dans ce qui suit nous commencerons par présenter quelques concepts de ce domaine et nous citons quelques méthodes dans leur ordre chronologique d'apparition, nous décrivons brièvement chaque approche en exposant un exemple illustratif de chacune d'elles. Ensuite, nous exposons nos deux contributions dans ce domaine. La première contribution consiste en un développement d'une méthode arrière de vérification. Elle est introduite dans la section 2. Tandis que la seconde, consiste en l'utilisation d'une méthode évolutionnaire pour la vérification de programmes. Elle est introduite dans la troisième section.

3.1. Etat de l'art

3.1.1. Model-Checking explicite énumératif

Les algorithmes de Model-Checking explicite énumératif sont basés sur l'exploration exhaustive de l'espace des états et des transitions du programme. Ils utilisent les différentes techniques de parcours et de recherche dans un graphe. L'état d'un programme peut être décrit comme l'information composée des variables, leur valeur et la valeur du compteur de programme représentant l'instruction en cours d'exécution Le problème de vérification des propriétés de sûreté d'un programme consiste à construire le graphe des états atteignables et à vérifier si l'état erroné, exprimant la violation de la propriété, est accessible dans ce graphe. Les model-checkers explicites fonctionnent selon un mécanisme de stockage d'états qui leur permet de rappeler les états antérieurement visités. Une fois un état feuille ou un état précédemment visité a été atteint, le model-checker peut revenir en arrière pour essayer les autres choix qui auraient pu être pris. Le Model-Checking explicite a été mis en œuvre dans plusieurs outils notamment Spin [Hol97] et Murphi [Dil96]. Ces deux outils ont eu un impact significatif, en particulier dans le domaine de la vérification des protocoles.

3.1.1.1 Problème d'explosion d'états

L'espace des états d'un programme étant généralement exponentiellement plus grand que la description du programme, ceci conduit donc au problème d'explosion d'états. C'est l'un des plus grands obstacles à l'application pratique de la vérification de modèle. Atténuer le problème d'explosion

d'états a donc été un axe principal de la recherche dans le domaine du Model-Checking de logiciel.

3.1.1.2 Quelques solutions au problème d'explosion d'état

Pour pallier au problème d'explosion d'état dans le cadre du Model-Checking explicite, la recherche a pris plusieurs directions. Tout d'abord, les techniques basées sur la réduction tentent de calculer des relations d'équivalence sur les comportements du programme, et d'explorer un candidat de chaque classe d'équivalence. Les premières techniques basées sur la réduction consistent en la réduction d'ordre partiel [Val92, KP92, God96]. La seconde famille de techniques est la réduction de symétrie, [CFJ93, ES96, ID96, SGE00]. Elle détermine et exploite les symétries dans le programme, et explore un élément de chaque classe de symétrie. Dans de nombreux exemples, tels que les protocoles paramétrés, les techniques à base de symétrie peuvent produire des réductions étonnantes de l'espace d'état [ID96]. Une autre classe de solutions est constituée des équivalences comportementales telles que la simulation ou bisimulation [BFH90, BG03]. Celles-ci tendent à construire un graphe quotient qui préserve l'accessibilité : il existe un chemin à partir d'un état initial à un état E dans le graphe original si et seulement si il existe un chemin menant à E dans le quotient. Ainsi, l'analyse d'accessibilité est effectuée sur le graphe quotient. Les techniques compositionnelles constituent une autre approche pour pallier aux problèmes d'explosion d'états. Elles permettent de réduire le problème de la vérification des propriétés de sûreté sur le programme initial en prouvant des propriétés sur les sous-programmes, de sorte que les résultats de vérification des sous-programmes peuvent être combinés pour déduire la sûreté du programme d'origine. L'approche Assume-garantee [Jon83, Sta85, AL95, HQR98, AHM$^+$98] est bien connue dans le raisonnement compositionnel. Dans cette approche, le comportement d'un composant se résume en une paire (A, G) de deux formules. A : une hypothèse sur l'environnement du composant, et, G : une garantie que le composant satisfera à condition que les entrées du composant vérifient A.

3.1.1.3 Exemples d'outils du Model-Checking explicite de programmes

Verisoft [God97] est un vérificateur de modèle explicite développé par les laboratoires Bell. La principale innovation dans Verisoft est d'introduire

l'idée de la recherche sans sauvegarder les états. Tout en ayant quelques inconvénients, cette idée augmente l'aptitude de l'outil à visiter plus d'états. Verisoft est capable d'explorer exhaustivement tous les entrelacements possibles des exécutions des processus. Il a été utilisé pour trouver plusieurs erreurs complexes dans des programmes concurrents de commutation en téléphonie [CGP02].

JavaPathFinder [HP00] (première version) : a été développé par les centres de recherche de la NASA. C'est un outil de vérification de programmes Java qui modifie la machine virtuelle Java pour mettre en œuvre la recherche systématique sur différents ordonnancements des tâches [HP00, VHB+03]. Cette approche offre de nombreux avantages significatifs. Tout d'abord, avec l'utilisation de la machine virtuelle Java, il est possible de stocker les états visités à l'aide de celle-ci, ce qui permet au vérificateur de modèle d'utiliser la plupart des approches de réduction (par exemple, la symétrie, ordre partiel,..) pour combattre l'explosion d'état. De plus, comme les états visités peuvent être stockés, le vérificateur de modèle peut utiliser diverses heuristiques de recherche. Finalement, des techniques comme l'exécution symbolique et l'abstraction sont utilisées pour calculer des données qui mènent le système dans des états différents de ceux précédemment visités de manière à obtenir un niveau élevé de couverture. Ces dernières caractéristiques ont été intégrées dans la seconde version de JavaPathFinder : SJPF[PR10]. JavaPathFinder a été utilisé avec succès pour trouver des erreurs subtiles dans plusieurs composants Java complexes développés à l'intérieur de la NASA [BDG+04].

3.1.2. Model-Checking Symbolique

Bien que les techniques énumératives capturent de la façon la plus fidèle l'essence même du Model-Checking qui est l'exploration exhaustive de l'ensemble des états du programme et des transitions, leur utilisation pratique est entravée par l'explosion de l'espace d'état. Cela a conduit à des recherches sur les algorithmes symboliques qui manipulent simultanément des ensembles d'états, plutôt que des états individuels. Ainsi, les model-checkers symboliques visitent un ensemble d'états simultanément. Cet ensemble d'états peut être représenté par des prédicats sur les variables d'états tels qu'un état appartient à l'ensemble si et seulement si le prédicat est satisfait dans ce même état. Les représentations succinctes jumelées aux manipulations efficaces de ces prédicats contribuent aux avantages du

Model-Checking symbolique relativement au Model-Checking explicite. Par exemple, la contrainte : $1 \leq x \leq 100 \wedge 1 \leq y \leq 5$ représente l'ensemble de tous les états sur {x, y} satisfaisant la contrainte. Ainsi, la contrainte représente implicitement la liste des 500 états qui auraient été énumérés dans le Model-Checking énumératif. En plus de fournir une représentation implicite des ensembles d'états, la technique doit également permettre d'exprimer certaines opérations sur les ensembles d'états directement sur la représentation. En effet, l'exécution de ces opérations en énumérant différents états d'une région symbolique annule l'avantage de la représentation symbolique. La puissance des techniques symboliques provient d'énormes progrès dans les performances des solveurs de contraintes et des solveurs de satisfiabilité [SS96, MMZ+01, ES03] ainsi que des diagrammes de décision binaires [Bry86].

3.1.2.1 Model-Checking symbolique basé sur les diagrammes de décision binaires

Les prédicats représentant des ensembles d'états peuvent être modélisés par des diagrammes de décision binaires : BDD[Bry86]. Les BDDs réduits et ordonnés sont similaires aux arbres de décision binaires à la différence que les sous arbres identiques doivent être fusionnés résultant en un graphe acyclique dirigé. Cet arbre a un nœud initial unique suivi de nœuds intermédiaires étiquetés avec des variables booléennes et finalement des nœuds terminaux étiquetés avec des '0' et des '1'. Chaque nœud intermédiaire possède exactement deux arcs dirigés du nœud lui-même à deux autres nœuds-fils : les deux arcs sont implicitement étiquetés avec '0' et '1'. Tester la satisfiabilité d'une fonction booléenne représentée par un BDD est plus simple qu'avec une représentation de la même fonction avec des formules propositionnelles ou des tables de vérité. L'utilisation des BDDs dans le Model-Checking a contribué à un essor considérable dans la vérification au début des années 1990[McM93]. L'un des privilèges associés à cette représentation est sa forme canonique : Pour la même fonction et le même ordre des variables, il existe un BDD unique représentant cette fonction. En plus, les BDDs offrent l'avantage d'exécuter des opérations de conjonction, disjonction ou négation avec une complexité de temps polynomiale. Cependant, l'un des obstacles majeurs de ce genre de Model-Checking réside dans la limitation des BDDs quant à la nature des variables manipulées, et l'ordre optimal de la considération de celles-ci. Un grand nombre de Model-Checkers symboliques basés sur les BDDs ont été

employés dans la vérification de circuits matériels. Ils représentent des ensembles d'état et souvent même la relation de transition par des BDDs.

Exemple d'outils basés sur les BDD

SMV [McM93] est l'exemple le plus marquant de cette famille d'outils. SMV a permis de vérifier des systèmes avec un nombre d'états considérable qui n'auraient pas pu être vérifiés auparavant.

3.1.2.2 Model-Checking symbolique borné

Biere et al [BCC+99] présentent une technique de Model-Checking symbolique borné basée sur les procédures de décision SAT. L'idée de base est de considérer des contre-exemples de taille déterminée k et de générer par la suite, une formule propositionnelle qui peut être satisfaite si et seulement si un tel contre-exemple existe. Par définition, le Model-Checking borné suppose que les erreurs peuvent être détectées dans un nombre limité de pas de recherche relativement au nombre d'états explorés. Cette approche n'est pas complète quant à la recherche de la totalité de l'espace d'états mais s'est avérée efficace dans la détection d'erreurs se localisant près d'un emplacement spécifié.

Exemples d'outils basés sur le Model-Checking borné

CBMC [KCY03], **F-Soft** [IYG+08], **Saturne** [XA05] sont des exemples d'outils basés sur le model-checking borné. Ils génèrent des contraintes et utilisent les solveurs de satisfiabilité pour les prouver. Par conséquent, l'efficacité des techniques dépend à la fois de l'efficacité des solveurs sous-jacents ainsi que des heuristiques appliquées pour garder la taille des contraintes raisonnable. CBMC et Saturne sont des outils mis en œuvre pour les programmes C. Tous les deux ont eu un certain succès dans l'analyse de gros morceaux de logiciel, y compris l'analyse de grandes parties du noyau du système d'exploitation Linux.

3.1.2.3 Model-Checking basé sur l'exécution symbolique

Le model-checking de programmes à l'aide de l'exécution symbolique a été introduit pour contribuer à la résolution du problème d'explosion d'états en plus de permettre l'utilisation des variables d'entrée dans des domaines non

bornés. Dans un premier temps, Khurshid et al. [KPV03] ont défini une traduction "source-à-source" pour instrumenter un programme, dans le but d'utiliser un model-checker standard à des fins d'exécution symbolique sans avoir recours à des outils dédiés. Le model-checker vérifie le programme en explorant automatiquement les différentes configurations du programme en manipulant les formules logiques sur les valeurs associées aux données de celui-ci. Suite à cela, les mêmes auteurs ont présenté un nouvel algorithme d'exécution symbolique qui permet la manipulation des éléments avancés de la programmation comme les structures de données dynamiques: Listes et arbres. Le model-checker vérifie le programme dûment instrumenté en utilisant ses propres techniques d'exploration de l'espace d'états. A chaque fois qu'une condition de chemin est mise à jour, elle est passée à une procédure de décision appropriée pour vérifier si elle peut être satisfaite.

3.1.3. Model-Checking basé sur l'abstraction

En dépit des résultats encourageants obtenus par les méthodes symboliques le problème d'explosion d'état persiste. Ceci est dû à la complexité croissante des logiciels à analyser. En effet, parallèlement à l'évolution des méthodes d'analyse des programmes, les logiciels ont eux aussi évolué de façon étonnante donnant ainsi naissance à de nouvelles difficultés. Cette complexité croissante est telle que l'abstraction de certains aspects non pertinents à l'analyse est vite devenue inévitable. L'abstraction d'un programme consiste à le représenter par un modèle ne représentant que les parties influençant l'analyse d'intérêt. Deux grandes familles d'abstraction ont vu le jour :

3.1.3.1 Interprétation abstraite

L'interprétation abstraite [CC77] est utilisée pour vérifier, de façon statique, les propriétés de sûreté des programmes. Un état erroné représente une localisation particulière du programme et les valeurs des différentes variables en cette localisation. L'idée est d'approximer l'ensemble des valeurs possibles des variables. L'interprétation abstraite est donc une théorie générale de l'approximation des sémantiques des programmes. Elle permet de formaliser l'idée qu'une sémantique est plus ou moins précise selon le niveau d'observation auquel on se situe. Ainsi, l'interprétation abstraite exprime toute sémantique comme un point fixe de fonctions

monotones. Elle se définit donc comme une théorie de résolution approchée d'équations de points fixes. Plusieurs domaines mathématiques abstraits ont été définis pour approximer les valeurs des variables ; les plus simples sont les intervalles.

Interprétation abstraite par intervalles

L'interprétation abstraite par intervalle [CC92] consiste à calculer en chaque point du programme un intervalle de valeurs pour chaque variable x (qui contient l'ensemble des valeurs prises par x en ce point). Pour cela, il suffit d'explorer tous les chemins d'exécution possibles en propageant les intervalles.

Exemple : Interprétation abstraite par intervalles sur un programme sans boucle

```
//  x∈[2,5]
y=2*x;                // y∈[4;10]
if (y<=4)
{  z=y-2; }           // z∈[2;2]
else { z=y+2; }   // z∈[7;12]
// z∈[2;12]
```

Pour les programmes avec boucles, le programme est parcouru jusqu'à ce que les intervalles obtenus soient des invariants. Dans le cas général, on n'obtient pas d'invariant en un nombre fini d'itérations, et on a donc recours à des opérateurs d'élargissement [CC77], qui, appliqués en tête des boucles, garantissent l'obtention d'invariants en temps fini. Les invariants obtenus à l'aide d'élargissement étant généralement trop larges pour montrer des propriétés intéressantes, ceux-ci sont réduits par un nouvel opérateur de rétrécissement.

Exemple d'outils basés sur l'interprétation abstraite

Astrée[Mau04] est un analyseur statique basé sur l'interprétation abstraite. Astrée peut découvrir automatiquement toutes les erreurs potentielles d'exécution d'une certaine classe de programmes en utilisant une approximation de la sémantique du langage C.

3.1.3.2 Abstraction des prédicats

Une autre famille d'abstraction consiste à ne considérer que quelques prédicats du programme parmi les prédicats des instructions conditionnelles. Ainsi, initialement, un ensemble minimal de prédicats est choisi, si la propriété peut être prouvée en ne considérant que ces prédicats le processus s'arrête sinon l'ensemble des prédicats est enrichi par d'autres prédicats. Un paradigme clé définissant cette technique a été défini : CEGAR : Counter-Examples Guided Abstraction-Refinment : Abstraction-Raffinement guidés par contre-exemples : [BR02, CGJ+00, GHK+06, IYG+05].

Approche CEGAR

Ce paradigme est basé sur la sur-approximation. Celle-ci consiste à représenter le programme par un modèle permettant plus de comportements que le programme initial. Ainsi, on tente de vérifier la propriété à partir d'une abstraction grossière du programme. Si la vérification réussit, alors on a la garantie que le programme concret satisfait la spécification. Si la vérification échoue, alors on produit un chemin qui viole la spécification abstraite dans le programme. Ce chemin peut être due soit à l'exécution du programme concret, fait que l'on pourra confirmer en exécutant ce chemin dans le programme original; ou bien survenir en raison de l'imprécision de l'abstraction. Dans ce dernier cas, l'abstraction du programme est affinée en tenant compte d'autres prédicats et le processus est réitéré. L'entrée de l'algorithme d'analyse de contre-exemple est un chemin dans le graphe de flot de contrôle se terminant dans la localisation de l'erreur. Le chemin représente un contre-exemple possible produit par l'analyse d'accessibilité. La première étape de l'algorithme construit une formule logique, appelée formule de trace, à partir du chemin, de telle sorte que la formule est satisfiable si et seulement si le chemin est exécutable par le programme concret. Ensuite, une procédure de décision est utilisée pour vérifier si la formule des traces est satisfiable. Si elle est satisfiable, le chemin est signalé comme un « vrai » contre-exemple de la propriété. Sinon, la preuve d'insatisfiabilité est exploitée pour inférer de nouveaux prédicats qui peuvent écarter le contre-exemple courant. Le paradigme CEGAR a constitué un tournant décisif dans le domaine de la vérification des programmes. En effet, plusieurs méthodes basées sur l'approche CEGAR ont été développées [HJM+04].

Exemples d'outils basés sur l'abstraction des prédicats

SLAM [BR02b] a été la première mise en œuvre de CEGAR pour les programmes C. Il a introduit des programmes booléens comme un langage intermédiaire pour représenter les abstractions. Dans un programme booléen, toutes les variables sont de type booléen. L'outil C2BP est utilisé pour générer un programme booléen à partir d'un programme C [BPR01]. L'entrée de C2BP est un programme C et un ensemble de prédicats ; la sortie est un programme booléen, où chaque variable booléenne correspond à un des prédicats entrés. Ce type d'abstractions est désigné par abstraction cartésienne. Un model checker symbolique : Bebop, pour les programmes Booléens a été mis en place [BR00a]. Slam a été utilisé avec succès pour la vérification de pilotes de périphérique de Windows [BBC+06]. Le projet Slam a introduit plusieurs idées clés dans la vérification de logiciel, y compris la généralisation de l'abstraction des prédicats en présence des pointeurs [BMM+01] et l'abstraction modulaire des prédicats [BMR05].

Blast [BHJ+07] construit un modèle abstrait sur lequel il vérifie des propriétés de sûreté. L'abstraction est construite à la volée seulement à la demande. L'explosion d'états est combattue en maintenant toujours le minimum d'informations nécessaires pour valider ou invalider la propriété et en réalisant des résumés de procédure [SP81, RHS95].

3.1.4. Aspects avancés de la programmation

3.1.4.1 Procédures

Beaucoup d'outils analysent les procédures en procédant à un 'in-ligning' où l'appel de la procédure est remplacé par le code de celle-ci en effectuant les substitutions nécessaires [CKL04]. Ainsi, chaque appel de la procédure nécessite une nouvelle analyse de celle-ci. Les résumés de procédures [GT07] sont une solution à cet inconvénient. Ils consistent à tenter de capturer l'effet de l'exécution de la procédure sur le programme appelant. Plusieurs méthodes, concrétisant cette idée, ont été développées. Les premières consistent à calculer une sur-approximation du comportement de la procédure [CCG+04]. D'autres approches calculent des résumés abstraits au moment de l'analyse, ainsi, seules les variables d'intérêt sont considérées. Cette approche est utilisée dans Blast. L'efficacité de la vérification est grandement améliorée en utilisant des résumés à appels

multiples, au lieu de résumés individuels qui ne peuvent être appliquées à d'appels spécifiques. Par exemple, il est typiquement plus efficace d'utiliser des abstractions relationnelles [CC92] qui peuvent décrire les résultats en termes d'entrées, au lieu de paires d'états d'entrée-sortie où chaque paire correspond à un appel différent. Un résumé relationnel qui spécifie que la sortie d'une procédure est supérieure à une entrée, est plus compacte et réutilisable qu'un résumé sous forme de tabulation qui spécifie par exemple, que si l'entrée est à 0, la sortie est 1. Ball, Millstein, et Rajamani [BMR05] ont introduit une méthode de calcul des résumés relationnels en utilisant l'abstraction des prédicats sur des prédicats contenant des constantes symboliques.

3.1.4.2 Structures de données dynamiques

Les structures de données dynamiques représentent l'un des défis les plus importants de la vérification des logiciels. La difficulté du raisonnement automatique sur les structures de données dynamiques découle de la nécessité de raisonner sur les relations entre les identificateurs syntaxiques et les emplacements mémoires auxquels ils réfèrent. En effet, deux expressions syntaxiquement distinctes e1 et e2 peuvent se référer au même emplacement mémoire, et donc la mise à jour de la mémoire en écrivant à l'emplacement e1 nécessite la mise à jour des informations sur le contenu de l'emplacement e2 syntaxiquement différent. L'analyse d'alias détermine si deux pointeurs sont une référence au même emplacement mémoire. Dans la littérature, il existe une grande variété de méthodes d'analyses d'alias [Ste96, Hin01, WL04]. L'hypothèse communément admise dans les analyses d'alias est d'abstraire chaque cellule mémoire allouée dynamiquement à partir du même emplacement du programme dans une cellule abstraite. Cela signifie qu'elles ne peuvent pas distinguer entre les différentes cellules d'une structure de données (par exemple, une liste chaînée) dont les cellules sont affectées en un même point du programme. Ceci conduit à l'imprécision. Pour formaliser l'analyse des programmes utilisant des pointeurs, la logique de Hoare a été étendue donnant naissance à la logique de séparation. La logique de séparation [Rey02] a été conçue pour permettre un raisonnement modulaire sur les programmes manipulant des pointeurs. La logique de séparation étend la logique de Hoare avec deux opérateurs, la séparation de la conjonction : notée '*' et la séparation de l'implication notée '-*'. Par exemple, une assertion de la forme A * B signifie qu'il y'a un ensemble de

cellules qui satisfont A et un ensemble disjoint de cellules qui satisfont B. Le principal avantage de cette logique est qu'elle permet de spécifier succinctement quelles parties de la mémoire sont touchées par un fragment de code donné. La logique de séparation a servi de base à plusieurs vérificateurs de logiciels basés sur l'interprétation abstraite. D'autres travaux ont également utilisé cette logique [DOY06, MBC+07, YLB+08].

3.1.5. Model-Checking et Test

Des améliorations et des hybridations des méthodes d'analyse de programmes sont réalisées de façon continue. Ces variantes constituent des tentatives parmi d'autres pour rendre cette analyse applicable à des programmes de taille et de complexité réelles. Parmi les hybridations les plus réussies : l'hybridation du model-checking avec le test. En effet, le model-checking et le test de logiciel ont toujours eu des forces et des faiblesses complémentaires. Ainsi, la tentative de combinaison des points forts du model-checking et du test est justifiée. Parfois le terme d'exécution symbolique dynamique est utilisé pour le test de programmes, indiquant que le programme est effectivement exécuté ; par opposition à l'analyse statique, dans laquelle un modèle du système est analysé sans exécution réelle du programme, comme c'est le cas dans la vérification. Dans [GHK+06, BNR+08], le model-checking de logiciels basé sur la méthode CEGAR et l'exécution symbolique dynamique ont été combinées pour rechercher simultanément un espace d'états abstrait pour, soit, produire une preuve de correction en utilisant l'abstraction des prédicats ou bien trouver un cas de test défaillant via les exécutions symboliques dynamiques sur des contre-exemples par le model checker. Si l'exécution symbolique dynamique trouve le contre-exemple infaisable, les techniques d'analyse habituelles de contre-exemples peuvent être utilisées pour affiner l'abstraction. L'algorithme combine les capacités de CEGAR pour explorer rapidement l'espace d'états abstraits, et de l'exécution symbolique dynamique, pour explorer rapidement les chemins de programme précis pour vérifier la faisabilité des contre-exemples. Plusieurs outils développés autour de cette idée ont fourni des résultats des plus encourageants. Parmi ceux-ci :

YOGI [ASR10] implémente l'algorithme Dash [GHK+06] qui effectue une vérification en combinant les tests et l'abstraction. Plusieurs expérimentations ont été réalisées sur les pilotes de Microsoft Windows Vista. Celles-ci ont montré la suprématie de Yogi relativement à Slam. Yogi traite des idées nouvelles pour gérer les alias de pointeurs et les appels de procédures. Il maintient simultanément une forêt de chemins de test et un graphe de régions d'abstraction du programme. Les tests sont utilisés pour trouver des erreurs et les abstractions sont utilisées pour prouver leur absence.

DART [GKS05]**.** L'idée de DART consiste en l'intégration des tests aléatoires et la génération de tests dynamiques à base de raisonnement symbolique. A partir d'une entrée aléatoire, un programme DART instrumenté calcule à chaque exécution un vecteur d'entrée pour la prochaine exécution. Ce vecteur contient des valeurs qui sont la solution des contraintes symboliques recueillies sur des prédicats au cours de l'exécution précédente. Le nouveau vecteur d'entrée tente de forcer l'exécution du programme grâce à un nouveau chemin. En répétant ce processus, une recherche dirigée tente de forcer le programme à balayer tous ses chemins d'exécution possibles. DART exécute le programme en cours d'analyse à la fois de façon concrète, avec des entrées aléatoires, et de façon symbolique par le calcul des contraintes sur les valeurs des emplacements de mémoire exprimées en termes des paramètres d'entrée.

3.2.BIMC: Backward Incremental Model-Checking [ATK13]

3.2.1. Principe et motivations

En dépit du grand nombre de méthodes et d'outils développés pour la vérification des propriétés de sûreté des programmes, ce domaine demeure très actif. En effet, sa problématique cruciale : explosion d'états et de chemins, n'a pas trouvé de solution ultime à ce jour. De ce fait beaucoup d'équipes de recherche persévèrent pour développer de nouvelles méthodes, améliorer les solutions existantes, et combiner les méthodes entre elles. Dans ce qui suit nous introduisons notre contribution à ce sujet. Nous avons développé une méthode d'exécution symbolique arrière pour la vérification de propriétés de sûreté pour le langage C : BIMC : Backward Incremental Model-Checking. Depuis son introduction, l'exécution symbolique a été utilisée principalement pour les tests de programmes. L'analyse en avant

procède en générant tous les chemins d'exécution possibles. Or, dans une analyse d'atteignabilité telle que la vérification de propriété de sûreté, ceci n'est pas nécessaire. En effet, l'objectif est de vérifier s'il existe au moins une exécution menant le programme à un état donné. De ce fait, l'exécution arrière nous parait la plus adéquate puisqu'elle permet de n'explorer que les parties en relation avec la propriété à vérifier. BIMC est une technique compositionnelle ascendante pour vérifier, de façon statique, les propriétés de sûreté des programmes. Un programme est représenté comme une hiérarchie de blocs. Les blocs sont explorés successivement, dans l'ordre inverse, jusqu'à ce que la propriété soit prouvée ou réfutée. Nous définissons une nouvelle méthode pour la modélisation des programmes: ASMA: A Separation Modeling Approach,. ASMA permet de séparer un programme en deux composantes: la première décrit toutes les opérations affectant la mémoire du programme : les valeurs des variables, nous l'appelons: Table des Variables (VT), la seconde est appelée Table de Contrôle (CT). Celle-ci décrit la structure de contrôle du programme d'une manière compacte. Cette approche de modélisation s'avère aussi simple qu'efficace. Elle permet une manipulation aisée des programmes. BIMC commence la vérification en démarrant d'un ensemble minimal P_0 contenant les conditions auxquelles est directement liée la propriété. L'ensemble P_0 est ensuite progressivement étendu à chaque fois que nécessaire. Le concept de plus faible précondition est appliqué sur les deux tables, VT et CT, d'une manière adéquate et progressive. Des investigations arrière démarrent à partir de chaque condition : Des calculs successifs des plus-faibles préconditions sont effectués pour retrouver les relations initiales sur les variables d'entrée qui permettent de rendre ces conditions simultanément satisfiables. BIMC permet de traiter des programmes contenant des appels de fonctions et des pointeurs. Notre approche est motivée par les points suivants:

1. Il s'agit d'une analyse orientée but : en effet, en plus du fait que l'ensemble P_0 initial est le plus petit ensemble de prédicats pouvant permettre de décider de la propriété, chaque élément de P0 est analysé de manière à pouvoir décider de sa satisfiabilité sans parcourir tout le chemin d'exécution.

2. L'analyse arrière est utilisée comme une technique d'abstraction, car elle omet tous les chemins et les prédicats qui ne sont pas nécessaires à l'analyse. En effet, celle-ci n'explore pas les chemins inutiles du programme, i.e. les chemins ne menant pas à la localisation

d'intérêt ni aux régions inutiles du code : par exemple, les affectations qui n'influencent pas la valeur de vérité du prédicat considéré.

3. Elle tire profit de la propriété de localité. En effet, le mode arrière permet d'utiliser, à chaque région du programme, les valeurs les plus récentes des variables. La propriété de localité a été prouvée être très efficace dans divers domaines de l'informatique. Soit l'exemple très simple suivant: supposons qu'une variable v a été attribuée plusieurs valeurs le long du programme et supposons que dans la localisation 1000 la valeur 0 à été assignée à v , et à la localisation 1010 nous devons suivre un prédicat contenant v si, dans une analyse arrière il suffit de remonter de 10 localisations pour trouver que v a la valeur 0, à l'inverse d'une analyse avant où toutes les affectations à la variable v de la localisation 1 à 1000 doivent être prises en considération.

Notre approche est compositionnelle. En effet, les fonctions sont modélisées indépendamment du programme appelant; chaque appel à une fonction est remplacé par des expressions qui résument l'effet de cet appel sur le programme appelant. Les pointeurs sont organisés en classes d'équivalence, chacune ayant un représentant. Toute modification effectuée sur un élément d'une classe donnée est exprimée sur le représentant de celle-ci.

3.2.2. Modélisation

3.2.2.1 Pré-processing

Avant de soumettre notre programme à l'analyse, une phase de prétraitement est nécessaire. Ceci est d'autant plus justifié que le langage C fait usage d'un grand nombre de formes syntaxiques non standards.
Nous préparons notre programme en appliquant les opérations suivantes:

1. Les instructions 'for' et 'do..while' sont remplacées par l'instruction 'while' équivalente.
2. L'instruction 'switch' est remplacée par l'instruction 'if then else'.
3. Les post et pré-incrémentation (décrémentation et) sont transformées en formes standards d'incrémentation et de décrémentation.

4. Les instructions de sortie : ‘Printf, puts,.. sont éliminées: elles n'ont aucun effet sur l'exécution.

Ainsi, après cette phase, un programme contient les instructions suivantes : l’affectation, les instructions conditionnelles, les répétitions et les appels de fonctions. Habituellement, les programmes sont représentés par des graphes de flots de contrôle, mais comme nous envisageons d'effectuer une analyse arrière ne couvrant que certaines parties choisies du programme et de développer une approche compositionnelle, il est primordial d'adopter une représentation des programmes qui facilite ces objectifs. Nous représentons les programmes par deux entités : VT la table des variables, et CT la table de contrôle.

3.2.2.2 Table des variables : VT

La table VT modélise toutes les modifications qui peuvent survenir dans la « mémoire » du programme. Elle modélise les déclarations des variables, les affectations, les entrées, les instructions return, et les appels de fonctions. Ces instructions sont toutes exprimées à l’aide des affectations et sont numérotées dans le même ordre de leur apparition dans le code source, avec des nombres entiers représentant leur localisation.

- La déclaration d'une variable *idf* avec un *type_idf* est modélisée par *idf = type_idf0*. Par exemple, *float x* est représenté par *x = float_x0* signifiant que *x* est une variable de type *float* dont la valeur est inconnue. Les déclarations sont numérotées par les localisations où elles sont effectuées.
- Les affectations sont représentées de la même façon que dans le programme source.
- Une instruction d'entrée : une lecture, a pour conséquence d'attribuer une valeur inconnue à une variable, ainsi, l'entrée d'une variable *v* est modélisée par : *v=$v*, où *$v* est interprétée comme une constante inconnue ou constante symbolique.
- Les fonctions prédéfinies et les appels aux librairies sont analysés en utilisant la notion de fonctions non interprétées : uninterpreted functions [BLS02]. Ce concept est utilisé dans beaucoup de méthodes d’analyse de programmes. Nous les représentons par des affectations de constantes spéciales inconnues dont les noms commencent par «£». Par exemple, un appel de la forme *v=rand (..)* ou *v = malloc (..)* est représenté par *v = £v*.

Ainsi, chaque ligne de VT est un triplet *(loc,var,exp)* où *loc* est la localisation de l'instruction dans le code source ; *var* est l'identificateur de la variable modifiée, et *exp* est la nouvelle expression de *var*.

3.2.2.3 Table de contrôle: CT

La table de contrôle décrit la structure de contrôle du programme. Elle définit les contraintes permettant l'exécution de chaque instruction de VT. Elle modélise les instructions conditionnelles et les boucles. Chaque entrée de la table CT est un quadruplet de la forme *(C,LT,LF,LE).*

- **Les instructions conditionnelles:** elles sont modélisées par le quadruplet *(C, LT, LF, LE)* où: *C* est la condition de l'instruction, *LT* est la localisation de la première instruction à exécuter si *C* est vraie; *LF* est la localisation de la première instruction à exécuter si *C* est fausse et *LE* est la localisation de la première instruction qui suit l'instruction conditionnelle. Une instruction conditionnelle sans la branche *Else* est représentée par *(C, LT, -1,LE).*

- **Les instructions répétitives:** une boucle est modélisée par *(C,LT,2,LE)* où: *C*,*LT* et *LE* ont la même signification que précédemment.

Par conséquent, les différents sous-ensembles de CT sont:

1. *ITE*: L'ensemble de tous les quadruplets ayant la forme *I = (C,LT,LF,LE)*, où *LF>0*. Dans ce cas: *Then(I) = [LT, LF[, Else(I)=[LF, LE[, C (I) = C* et *Begin (I) = LT*. La notation *[l1,l2[* représente l'intervalle des localisations de *l1* (inclus) jusqu'à *l2* (exclu).
2. *IT*: L'ensemble de tous les quadruplets de la forme *I = (C, LT, -1,LE). Then(I), C (I)* et *Begin (I)* sont définis comme précédemment.
3. *LOOP*: L'ensemble de tous les quadruplets de la forme *I=(C,LT,-2,LE). Body(I) = [LT,LE[* et *Begin (I) = LT.*

Les fonctions sont modélisées indépendamment du programme appelant, de la même façon que les programmes.

Exemple de modélisation d'un programme : Dans la figure 3.1, nous présentons un programme avec sa modélisation. Nous ne représentons qu'une seule déclaration, la même syntaxe est utilisée pour toutes les autres déclarations.

Programme *Prog*

```
1 : int x,y,z,t;
5 :scanf("%d",&z)
6 :scanf('%d',&x)
if (z==x)
   { 7: t=x+1;
     8: y=x-1;
   }
else
    { 9 : t=x-1;
      10:y=x+1 ;
    }
11 :t=t+y;
if (z<3)
    12 : t=z ;
else { 13 :t=-z;
         if (t==-1)
           14 :y=0;
        else  15 : y=1;
```

Loc	Var	Exp
1	X	int_x0
5	Z	$z
6	x	$x
7	T	x+1
8	Y	x-1
9	T	x-1
10	Y	x+1
11	t	t+y
12	T	Z
13	t	-z
14	y	0
15	y	1

	C	LT	LF	LE
1	z=x	7	9	11
2	Z<3	12	13	16
3	t=-1	14	15	16

Figure 3.1. Exemple de modélisation de programme

3.2.2.4 Propriété

Comme il est d'usage dans les méthodes de vérification des propriétés de sûreté, la propriété est exprimée par l'instrumentation du code source. Elle sera donc représentée par une localisation dans le code source qui n'est accessible que si la propriété de sûreté est violée. Ainsi, par exemple, la propriété de sûreté « il n'y a aucune exécution pour laquelle la valeur de la variable *x* est négative à une localisation *l* donnée » est exprimée dans le code par *if (x<0) l : <inst>*. Il s'agira donc de prouver qu'il n'existe aucune exécution atteignant *l*.

3.2.3. Description détaillée de la méthode

Avant de détailler notre approche, introduisons d'abord quelques définitions.

3.2.3.1 Définitions préliminaires

Définition 1 (Programme): Nous définissons un programme *P* par: *P={(VTi, CTi). i = 1 .. Nfunc}* où *Nfunc* est le nombre de fonctions dans le programme. On suppose dans un premier temps que *Nfunc = 1*. Donc, *P = (VT, CT).* Les fonctions seront introduites dans la section 3.2.5.

Définition 2 (Bloc): Chaque élément de CT est appelée un bloc. Une localisation appartient à un bloc si elle est située dans l'un de ses intervalles. Une localisation appartient à CT s'il ya un bloc de CT auquel elle appartient. Soient B1 et B2 deux blocs. Nous disons que B1 contient B2 ou $B2 \subset B1$ si toutes les localisations appartenant à B2 sont dans B1. Un bloc de base est un bloc qui ne contient aucun autre bloc.

Définition 3 (Exécution): Soit P = (VT, CT) un programme. Une exécution est toute substitution permettant de remplacer chaque constante symbolique, de P, de la forme $v par une valeur de même type.

Définition 4 (chemin d'exécution d'une localisation): Un chemin d'exécution P_i d'une localisation L, noté par P_{iL}, est un ensemble de blocs de CT tels que, si leurs localisations sont suivies en séquence, elles conduisent à L. Dans un chemin d'exécution donné, le prédécesseur (direct) d'un élément E est l'élément qui vient juste avant. Un prédécesseur indirect de E est un élément venant avant E. La longueur du chemin d'exécution P_{iL} est le nombre d'éléments de P_{il}. L'ensemble de tous les chemins qui mènent à L est noté P_L.

Définition 5 (satisfiabilité): Soient :

- IC: L'ensemble des constantes symboliques représentant les variables d'entrée (débutant par $)
- C: L'ensemble des autres constantes manipulées par le programme.
- Var: L'ensemble des variables du programme.

Soit F une formule sur l'ensemble $V = C \cup IC \cup Var$. F est satisfiable si et seulement si l'une des deux conditions suivantes est vérifiée:

1 - F est valide. Ou
2 – Il existe une substitution des éléments de IC tels que F est vraie.

La négation du point (2) est la suivante: Pour toute substitution des éléments de l'ensemble IC, F est fausse ou inconnue. La valeur de vérité inconnue est dû au fait que, dans F, nous disposons des éléments de l'ensemble Var des variables qui ne sont pas connues (et qu'on ne peut pas substituer). Nous effectuons donc la distinction entre les deux cas:
Si pour toute évaluation des constantes de l'ensemble IC, F est fausse alors F est non satisfiable sinon nous affirmons que nous ne pouvons pas décider.

Exemples

F1 = (x+1> x): F1 est satisfiable, car elle est valide

F2 = ($x+ $y=0): F2 est satisfiable. $x et $y sont des éléments de IC, nous pouvons attribuer les valeurs: $x=$y=0 pour que F2 soit vraie.

F3 = ($x + y = $x)$\wedge$(y$\neq$ 0), F3 n'est pas satisfiable: Pour toute valeur de $x, F est fausse.

F4 = (x + $y +1 = 1):

1 - F4 n'est pas valide.

2 - Nous ne pouvons pas attester que F est fausse pour toutes les valeurs de $y.

3 - Il n'y a pas d'évaluation de $y telle que F soit vraie

Nous concluons à travers ces trois points que nous ne pouvons pas décider si F4 est satisfiable ou non.

Définition 6 (initialisation de boucle): Soit B un bloc de CT de la forme (C, LT,-2,LE), nous appelons initialisation de la boucle B par rapport au chemin P_i nous notons LI_{iB}, l'ensemble de tous les éléments de VT de la forme: l_k : v= exp tel que:

1 - v est une variable qui apparaît dans C.

2 - $l_k \in P_i$

3 - Il n'existe aucune localisation l_k' tel que: l_k': v = exp' et $l_k' \in P_i$ et $l_k' > l_k$.

3.2.3.2 Description de la méthode

Dans cette section, nous décrivons la technique dans l'ensemble, les détails concernant chaque étape sont donnés dans les sections suivantes. Dans un premier temps, nous ne considérons que les programmes ayant une fonction unique. Par conséquent, un programme Pg est représenté par une table de variables VT et une table de contrôle CT. La propriété de sûreté est représentée par une localisation erronée L. Le but est de vérifier s'il existe une exécution du programme Pg permettant d'atteindre L. Les étapes suivantes sont effectuées (voir figure 3.2) :

1. En premier lieu, l'ensemble P_0 ne contenant que les conditions qui astreignent directement la localisation L est construit. P_0 est construit à partir de tous les éléments de CT contenant L. Ces éléments sont insérés dans P_0 dans le même ordre de leur apparition dans CT. Chaque élément de P_0 est de la forme (Ci, [Si,Sj[) signifiant que pour accéder à l'ensemble des localisations de Si à Sj : [Si,Sj[, il est nécessaire que la condition Ci soit vraie. Chaque élément inséré dans P_0 est retiré de CT. On ajoute au début de P_0 : (True, [1,Begin(I)[) où I est le premier élément de P_0, pour représenter le fait qu'avant le premier bloc, toutes les localisations sont supposées automatiquement accessibles.

2. P_0 est introduit dans la procédure d'analyse de faisabilité pour vérifier s'il représente un chemin faisable. Ce qui signifie qu'il existe une exécution pour laquelle toutes les gardes Ci (i=1 à n) de P0 soient vraies simultanément. La procédure d'analyse de faisabilité utilise le concept de plus faible précondition pour générer, pour chaque élément Ei = (Ci, [Si, Sj[) de P_0, un nouveau prédicat Ci' dont la valeur de vérité avant le premier élément de P_0 entraîne celle de Ci avant Ei. Au lieu de vérifier chaque condition Ci séparément (n appels à l'assistant de preuve), nous vérifions la formule $F=\wedge_{i=1,n} Ci'$ (un seul appel à l'assistant de preuve). Voir la figure 3.2. Trois cas sont possibles concernant la satisfiabilité de F :
   - **F est satisfiable :** L'ensemble P_0 est suffisant pour prouver que la localisation L est accessible. Le processus se termine, l'ensemble des assignations rendant F vraie est présenté comme contre-exemple.
   - **F n'est pas satisfiable :** P_0 est suffisant pour prouver que L n'est pas accessible. Le processus est arrêté. Le programme vérifie la propriété de sûreté.
   - **Nous ne pouvons pas décider :** l'ensemble P_0 ne suffit pas pour décider de l'accessibilité de la localisation L. Dans ce cas, P_0 doit être étendu avec des éléments de CT qui n'ont pas encore été examinés. Ainsi, puisque notre technique est basée sur des investigations en amont, donc, des éléments de CT situés avant P_0 sont pris en considération. Nous ne considérons pas ces éléments tous ensemble, mais progressivement, en commençant par le plus proche et le plus externe dans le cas de blocs imbriqués, et seulement si nécessaire.

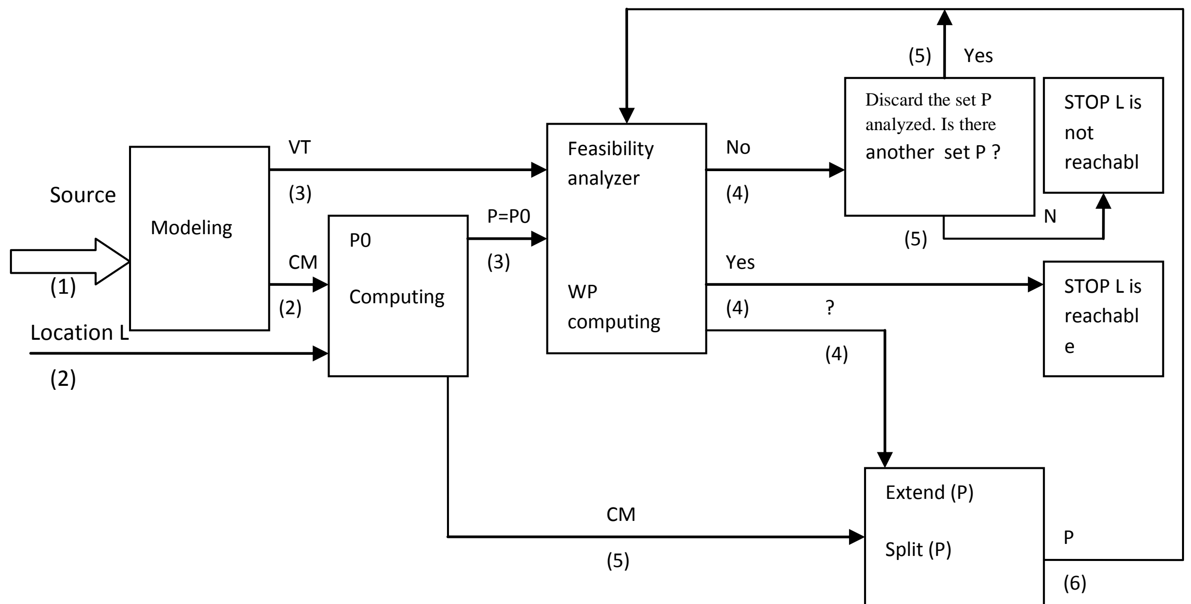


Figure 3.2. Architecture de BIMC

3. Soit Pi l'ensemble obtenu en étendant P, (dans un premier temps P0), par l'élément le plus externe de CT situé avant P. Cet élément représente une boucle ou une instruction conditionnelle située avant L. Ainsi, dans le calcul de la plus faible précondition, nous devons considérer toutes les exécutions possibles. Cela peut conduire à diviser l'ensemble Pi en deux ensembles Pi1 et Pi2 chacun représentant un chemin possible.

4. Les étapes 2,3 et 4 sont répétées avec Pi (ou Pi1 et Pi2) ayant le même rôle que P0. Nous ne calculons les plus faibles préconditions que des éléments ayant la forme (Ci,[Si,Sj[) par rapport à tous leurs prédécesseurs. En effet, la formule F ne contient que les éléments de la forme (Ci,[Si,Sj[), puisque ce sont ces éléments qui constituent les blocs gardés contenant la localisation. Dans le cas où il existe plus d'un ensemble Pi, le deuxième point de l'étape 3, c'est à dire si la formule F n'est pas satisfiable, ne conduit pas à abandonner tout le processus, mais à écarter seulement le chemin Pi considéré.

3.2.3.3 Algorithme BIMC

Algorithme 1: BIMC

1: Entrées: Pg=(VT,CT) ; L : une localisation.
2: Sorties: Eventuellement, une exécution E_i et un chemin d'exécution P_i
3: Initialisations: i=0
4: Calculer l'ensemble intial P_0 ; Pths={P_0};
5: Pour chaque élément de Pi de la forme (C_k,[Sm,Sn[) calculer sa plus faible Précondition C_k'
6: $F = C_1' \wedge C_2' \wedge \ldots . \wedge C_n'$. Vérifier la Satisfiabilité de F.
7: If (F est satisfiable)
8: Then Exit: L est accessible,
L'ensemble des assignations rendant F True est une exécution
P_i est un chemin d'exécution.
9: Else If (F n'est pas satisfiable)
10: Then Pths=Pths-{P_i} ; **If** Pths= $\varnothing$ **then** Exit: L n'est pas accessible
11: **else** P_i:=next_elt(Pths);
12: **Goto 5**
13: Else If (Extension)
14: Then Extend(Pi);
15: Goto 5
16: Else Split(P_i, P_{i1}, P_{i2})
17: Pths=(Pths-{P_i})$\cup${P_{i1},P_{i2}}
18: P_i=next_elt(Pths);
19: **Goto 5**;
20: END.

Figure 3.3. Algorithme BIMC

3.2.3.4 Construction de l'ensemble Initial d'investigation P_0

La construction de l'ensemble P_0 utilise la table de Contrôle. CT est maintenue dans le même ordre d'apparition de ces éléments dans le code source. L'algorithme calculant l'ensemble P_0 est donné par la figure 3.4.

Algorithme 2: Calcul de P_0

1: Entrées: Une localisation L ; CT.
2: Sorties: P_0 : L'ensemble des blocs gardés contenant L.
3: Initialisation: $P_0 = \emptyset$,i=1;
4: While(True)
5: Do Begin
6: If fin (CT)
7: Then exit
8: Else let E = CT[i];
9: **If**($L \in E$)
10: **Then** CT=CT-{E};
11: **if** ($P0=\emptyset$) **then** $P0=P0 \cup$ {(True,[1,Begin(i)[)};
12: **Case 1**: $E \in ITE$:
13: **If** $L \in Then(E)$
14: **then**$P0=P0 \cup$ {(C(i),Then (i))}
15: **else** $P0=P0 \cup$ {($\neg$C(i), Else(i))}
16: **Case 2**: $E \in IT$: $P0=P0 \cup$ {(C(i),Then (i))}
17: **Case 3**:$E \in LOOP$:$P0=P0 \cup$ {$(C(i),Body(i))^{+}$}
18:i=i+1; **End.**

Figure 3.4. Algorithme de calcul de l'ensemble P0 des blocs contenant la localisation L.

- L'insertion de (True,[1,Begin(i)[) au début de P0 exprime le fait que nous faisons abstraction de toutes les contraintes qui sont avant le début de P0. Donc, nous supposons dans un premier temps que nous accédons à P0 sans aucune contrainte, ou de façon équivalente, avec une contrainte : True. Cette hypothèse sera modifiée dans le cas d'extension de P0.
- La succession des blocs de P0 constitue le chemin de la localisation 1 à L à condition que la garde de chaque élément de P0 soit vraie.
- Lors de l'insertion d'un élément dans P0, les bornes des intervalles voisins sont ajustées de façon à ce que le début de l'élément nouvellement inséré coïncide avec la fin de l'élément précédent sauf dans le cas où l'élément précédent représente une boucle.

Exemples : Dans le programme Prog de la figure 3.1 :

- L=6 : $P0=\emptyset$
- L=10: P0={(True, [0,7[) ;($z \neq x$,[9,11[)}
- L=14: P0={(True, [0,12[) ;($z>=3$,[13,14[); ($t=-1$, [14,15[)}.

3.2.3.5 Analyse de faisabilité

L'analyse de faisabilité consiste à vérifier si un ensemble P représente un chemin d'exécution faisable conduisant à L. Les éléments de P sont explorés successivement dans l'ordre inverse. Pour chaque élément de la forme (Ci,[Si,Sj[), les plus faibles préconditions de Ci relativement à ses prédécesseurs directs et indirects sont calculées jusqu'à ce qu'elles atteignent une valeur constante: True ou False, ou le calcul atteint le premier élément de P. Par conséquent, à la fin de cette phase, nous avons pour chaque élément de P ce que devrait être sa garde juste avant le premier élément de P. Ainsi, nous vérifions si toutes les gardes du chemin P peuvent être satisfaites simultanément. De cette manière, au lieu de vérifier chaque condition séparément, nous vérifions toutes les conditions en même temps au même endroit (au début de P). Le schéma de la figure 3.5 illustre cette idée sur le programme Prog de la figure 3.1.

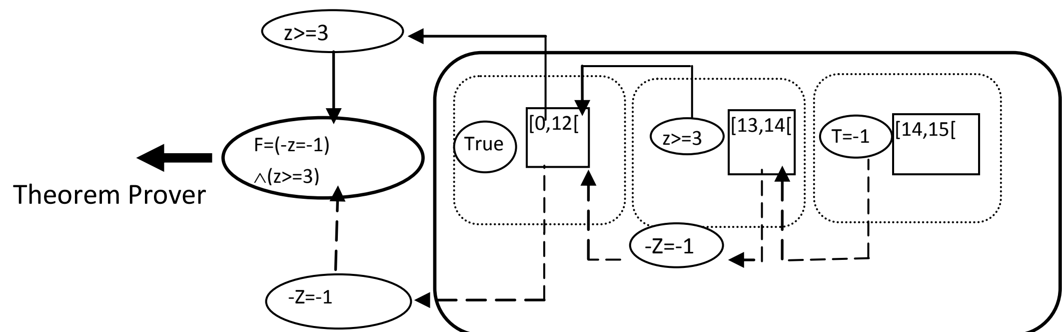


Figure 3.5. Processus d'analyse de faisabilité (L=14)

3.2.3.6 Calcul des plus faibles préconditions

3.2.3.6.1 Plus faible précondition d'un prédicat relativement à un intervalle

Nous avons par définition WP(v=exp, C) = C[exp / v]. C'est à dire C avec toutes les occurrences de v remplacées par exp. Dans la suite, nous notons WP(Si, C) la plus faible précondition du prédicat C relativement à l'instruction ayant la localisation Si dans la table des variables. La plus

faible précondition d'un prédicat C relativement à un intervalle [Si, Sj[, notée WP([Si,Sj[,C), est calculée de façon classique: WP([li,lj[,P)=WP([li,lj-1[, WP(lj-1,P)). Toutefois, nous devons vérifier pour chaque localisation Sk∈[Si,Sj[, si elle appartient à CT. Dans ce cas, le calcul de WP est interrompu jusqu'à ce que nous vérifions d'abord si la formule F obtenue par la conjonction des prédicats de tous les éléments de P est satisfiable. Si nous pouvons décider, alors le processus s'arrête, ce qui signifie que les éléments de P sont suffisants pour affirmer que la localisation est accessible ou non, sinon l'ensemble P doit être étendu.

3.2.3.6.2 Plus faible précondition d'un prédicat relativement à un élément de l'ensemble P

Introduisons d'abord les notations suivantes : Soit L = {l_1, l_2, ... l_k,} un ensemble de localisations tels que $l_1 < l_2 < ... < l_k$ et C un prédicat, nous notons CL = WP (l_1, l_2, ..., l_k, C) où la notation l_1,l_2,…l_k représente la séquence d'instructions localisées en l_1 ;l_2 ;…l_k.
La plus faible précondition du prédicat Cd relativement à un élément E de l'ensemble P, notée WP (E, Cd) est réduite à son calcul par rapport à l'intervalle adéquat sous la condition adéquate. Appelons E1 l'élément contenant Cd, et soit Sb = Begin (E1). Soient également, C_{nLI} D_{nLI} et B_{nLI} respectivement les plus faibles préconditions de C_n, D_n et B_n relativement à l'ensemble d'initialisation de la boucle LI. Nous avons cinq cas concernant la forme de E présentés sur la figure 3.6 :

Algorithme 3: WP(E,Cd)

- **Cas 1 :** E de la forme (C,[Si,Sj[): WP(E,Cd) = WP([Si,Sj[,Cd)
- **Cas 2:** E de la forme(C,LT,LF,LE) avec LF>0
  $WP(E,Cd)=C \wedge WP([LT,LF[,Cd) \vee \neg C \wedge WP([LF,LE[,Cd)$
- **Cas 3 :** E de la forme (C,LT,-1,LE) :
  $WP(E,Cd)=C \wedge WP([LT,LE[,Cd) \vee \neg C \wedge Cd$
- **Cas 4:** E de la forme (C,[LT,LE[)* : $C_0=C$; $CD_0=Cd$; n=1;
  While (True)
  Do { $CD_n=WP([LT,LE[,CD_{n-1})$; $C_n=WP([LT,LE[,C_{n-1})$;
  If $(Cn_{LI}=False) \vee (CD_n=CD_{n-1})$ {$WP(E,Cd)=CD_{nLI}$;exit}
  n=n+1; }
- **Cas 5 :** E de la forme $(C,[LT,LE[)^+$: $C_0=C$; $CD_0=Cd$; n=1;
  While (True)
  Do { $B_n=WP([LT,Sb[,CD_{n-1})$; $CD_n=WP([Sb,LE[,B_n)$;
  $C_n=WP([LT,LE[,C_{n-1})$;
  If $(C_{nLI}=False) \vee (B_n=B_{n-1})$ { $WP(E,Cd)=B_{nLI}$; exit }
  n=n+1; }

Figure 3.6. Calcul de la plus faible précondition relativement à un élément de l'ensemble P.

- **Cas 1 :** La plus faible précondition est calculée sans tenir compte d'aucune alternative, puisque le chemin [Si,Sj [doit être suivi.

- **Cas 2 et 3 :** Il est nécessaire de considérer toutes les alternatives selon la valeur de vérité de la condition C. Ainsi, dans le cas 2, la plus faible précondition doit être calculée soit en [LT,LF[ou [LF,LE[en fonction de la valeur de vérité de C. Le cas 3 présente deux situations possibles: soit la condition C est vraie et la branche THEN est exécutée ou C est fausse et, par conséquent, l'instruction If n'a pas d'effet sur la condition Cd.

- **Cas 4 :** Ce cas représente le calcul de WP relativement à une boucle avec la condition Cd à l'extérieur (i.e. située après la boucle). Ainsi, la boucle peut être exécutée zéro ou plusieurs fois. Dn représente l'expression de Cd après n itérations, et Cn est l'expression de C après n itérations $(n>=0)$. C_{nLI} est la valeur de la condition C après n itérations. Par exemple, si nous considérons l'exemple d'une boucle avec la condition suivante: $C=(i<j)$, l'ensemble d'initialisation {i=1 ; j=10} et que chaque itération incrémente la variable i de 1, sans modifier la variable j. Par exemple, pour n=3: $C3=(i+1+1+1)<= j$ et $C_{3LI}=WP(i=1,C3)=(4<10)$. Le processus de calcul des plus faibles préconditions se termine si la condition C de la

boucle devient fausse ou Dn atteint un point fixe. i.e. Cd n'est pas modifiée par la boucle.

- **Cas 5 :** Ce cas représente le calcul de WP relativement à une boucle avec Cd à l'intérieur de celle-ci. Donc, la boucle doit être exécutée au moins une fois. CDn représente l'expression de Cd à la fin de l'itération n et Bn l'expression de Cd au début de E1 après n itérations. Le processus de calcul de la plus faible précondition se termine si la condition de la boucle devient fausse ou Bn atteint un point fixe.

3.2.3.6.3 Algorithme d'analyse de faisabilité

L'algorithme effectuant l'analyse de faisabilité est donné par la figure 3.7:

Algorithme 4: Feasible(P,F,ExtendPos))

Inputs: A path P having N elements : E_1,....E_N
Outputs: A Formula F, ExtendPos
Initializations: F=True**,** Explore=False;
For each element $E_{k\ (k>1)}$ having the form (C_k,[Si,Sj[)
Do Begin C_k'=C_k ; r=k-1;
    While(r>= 1) and (Explore=False)
    Do Begin C_k'= WPC(E_r ,C_k') ; r=r-1; **End**
    F=F$\wedge C_k$';
    If (Explore=True) **Then** ExtendPos=r; Exit;,
  End;

Figure 3.7. Algorithme d'analyse de faisabilité.

3.2.3.7 Extension de chemin

Comme nous l'avons précédemment expliqué, si nous ne pouvons pas décider de la satisfiabilité de la propriété, à l'aide de P (P0 ou un autre Pi généré après) un nouvel élément doit être ajouté à P. Cette situation se produit lorsque les deux raisons suivantes sont établies simultanément:

1. Dans le calcul de WP ([Sm, Sp[, C), nous rencontrons une certaine localisation Sk qui appartient à un élément E de CT. Par conséquent, l'accessibilité de Sk est soumise à des contraintes supplémentaires.
2. Nous ne pouvons pas décider si la formule F est satisfiable (étape 2 de la méthode).

L'ajout d'un nouvel élément au chemin P est effectué par son insertion dans P et son retrait de CT.

Algorithme 5: Extend(P)

Inputs: A path P having N elements : E_1,..E_N,;ExtendPos ; CT
Outputs: The path P with N+1 elements
t=ExtendPos; let E be the element of CT to insert
If (t=1) **Then** E_1= (True,[1,Begin(E)[)
If E of the form (C,LT,-2,LE) Then E= (C,[LT,LE)*
Insert (E) at the position (t+1) ;
End;

Figure 3.8. Algorithme d'extension d'un chemin

3.2.3.8 Fractionnement de chemin

Lors de l'extension du chemin P, nous considérons d'abord l'élément ajouté, E, comme une entité indissociable. Cependant, le calcul de la plus faible précondition relativement à E peut être sans apport, donc, nous devons explorer à l'intérieur de E. Cette situation peut entraîner le fractionnement du chemin P. En effet, par exemple dans le cas d'un élément E de la forme (C,LT, LF, LE) si on doit explorer à l'intérieur de E, nous devons étendre le chemin en deux directions opposées: dans [LT, LF[avec un élément : E1, et dans [LF,LE[avec un autre élément E2. Or puisque l'ensemble P représente un chemin unique linéaire, on ne peut pas ajouter deux éléments représentant des portions de chemin opposées. Ainsi, nous créons deux copies de P: P1 et P2. Nous étendons P1 et P2 respectivement par E1 et E2. L'algorithme réalisant le fractionnement de chemin est donné par la figure 3.9 :

Algorithme 6: Split(Pi,Pi1,Pi2)

Inputs: A path Pi ; ExtendPos
Outputs: Two paths: Pi1 and Pi2
Create two copies Pi1 and Pi2 of Pi; t=ExtendPos;
Let E1, and E2 be the CT elements to add

- **Case 1:** E_t=(C,LT,LF,LE):
  In P1 transform E_t in (C,[LT,Sb1[); insert E1 after E_t.
  In P2 transform E_t in (¬C,[LF,Sb2[);insert E2 after E_t.

- **Case 2:** E_t=(C,LT,-1,LE):
  In P1 transform E_t in (C,[LT,Sb[) ; insert E1 after E_t.
  In P2 Remove E_t.

- **Case 3:** E_t=(C,[LT,LE[)*:
  In P1 transform E_t in (C,[LT,LF[)$^+$; insert E1 after E_t
  In P2 Remove E_t.

End;

Figure 3.9. Algorithme de fractionnement de chemin

Dans le premier cas puisque nous devons explorer à l'intérieur de l'élément Ei, donc nous devons distinguer entre les deux directions opposées : [LT,LF[avec la condition C, ou [LF,LE[avec la condition ¬C. Pour le second cas, dans P2, si la condition est fausse, l'instruction conditionnelle n'a aucun effet et par conséquent nous retirons E. Le même raisonnement est appliqué au cas 3 dans P2.

3.2.4. Exemples illustratifs

Afin d'élucider chaque étape de notre méthode, nous présentons quelques exemples illustratifs. Dans tous les exemples, nous avons représenté une seule déclaration, les autres déclarations sont représentées de la même façon.

- **Exemple 1**

Dans le programme Prog de la figure 3.1, vérifions la propriété suivante : Il n'y a aucune exécution permettant d'atteindre la localisation 14.

1. Calcul de P_0 : P_0 = {(True, [1,12[), (z>=3, [13,14[), (t=-1, [14,15[)}

2. Analyse de faisabilité:

2.1. Les plus faibles préconditions:

WP ([13,14 [, t = -1) = (-z = -1);

WP ([0,12[,-z = -1)= ? WP(11,-z=-1)= -z=-1;

WP(10,-z=-1) 10∈ CT ⇒ Stop.

WP([0,12[,z>=3)= ? WP(11,z>=3)= z>=3

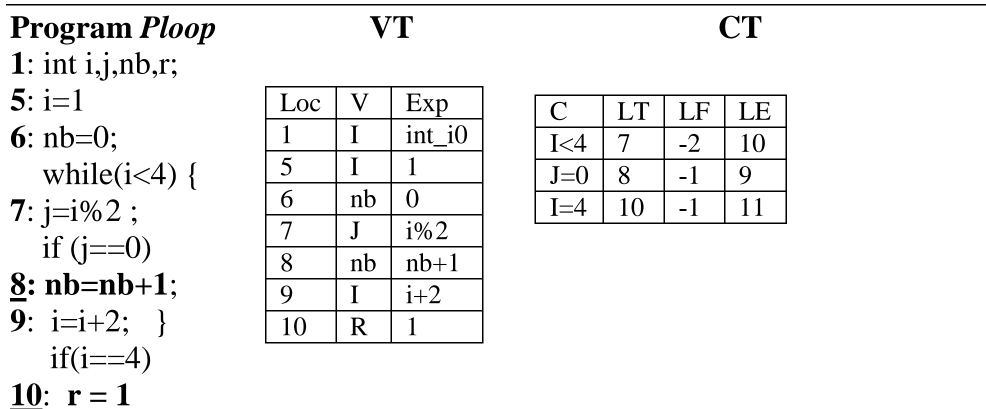
WP(10,z>=3) 10∈ CT ⇒ Stop.

2.2. Calcul de la formule : F = (-z =-1)∧(z > = 3)

2.3. Satisfiabilité de F: F = (z = 1) ∧ (z> = 3), il est évident que quelque soit la valeur de la variable z, la formule F est fausse. D'où F n'est pas satisfiable. Ainsi, le processus se termine.

3. Conclusion : Il n'existe aucune exécution permettant d'atteindre la localisation 14. Dans cet exemple, P_0 est suffisant pour prouver la propriété.

- ## Exemple 2 : Boucle et extension de P_0



Program *Ploop*

```
1: int i,j,nb,r;
5: i=1
6: nb=0;
   while(i<4) {
7: j=i%2 ;
   if (j==0)
8: nb=nb+1;
9:  i=i+2;  }
    if(i==4)
10:  r = 1
```

VT

Loc	V	Exp
1	I	int_i0
5	I	1
6	nb	0
7	J	i%2
8	nb	nb+1
9	I	i+2
10	R	1

CT

C	LT	LF	LE
I<4	7	-2	10
J=0	8	-1	9
I=4	10	-1	11

Figure 3.10. Exemple 2: Programme avec boucle et extension de chemin

Vérifions, dans un premier temps, si l'emplacement 10 est accessible:

1. Calcul de P_0: P_0 = {(True, [1,10 [), (i = 4, [10,11 [)}

2. Analyse de faisabilité:

2.1. Les plus faibles préconditions:

WP ([1,10 [, i=4): WP (9, i=4) =? Stop car 9∈(i <4,7, -2,10).

2.2. Calcul de la Formule: F = True∧(i=4)

2.3. Satisfiabilité de F: Nous ne pouvons pas décider ⇒ P_0 doit être étendu.

3. Extension de P0: P1 = {(True, [1,7 [), (i <4,[7,10[)*, (i = 4, [10,11 [)}.

4. Analyse de faisabilité: Soit E=(i <4,[7,10[)*, et LI = {5: i = 1} l'ensemble d'initialisation de la boucle.

4.1. Les plus faibles préconditions:

WP (E,i=4): Cas n ° 4 de la figure 3.5. Le calcul de Dn et Cn à chaque itération nécessite l'insertion du bloc (j=0,8,9), puisqu'elle n'a aucun effet sur Dn et Cn, nous l'omettrons..

n=0: C0 = (i <4); D0 = (i = 4)

n=1: D1 = WP ([7,10 [, D0)=(i +2=4);

C1=WP([7,10[,C0)=(i+2<4);

$C1_{LI}$ = (1 +2 <4) ≠ false, D1 ≠D0

n=2: D2=WP([7,10[, D1)=(i+2+2=4);

C2=WP([7,10[,C1)=(i+2+2<4);

$C2_{LI}$=(1 +2+2<4)=False=> Stop.

WP(E,i=4)=$D2_{LI}$=(1+4=4)=False.

4.2. Calcul de la Formule: F = True∧False=False.

4.3. Satisfiabilité: La formule F n'est pas satisfiable.

5. Conclusion: Il n'existe aucune exécution permettant d'atteindre l'instruction à la localisation 10.

Vérifions maintenant, dans le même programme Ploop, s'il existe une exécution permettant d'atteindre la localisation 8 :

1. Calcul de P0: P0 = {(True, [1,7 [), (i <4, [7,10 [)$^+$; (j = 0, [8,9 [)}.

2. Analyse de faisabilité: Nous appelons E=(i <4, [7,10[)$^+$

2.1.Les plus faibles préconditions: WP(E, j=0): cas 5 de la figure 3.5

n=0 : C0 = C = (i <4); A0 = (j = 0)

n=1: B1=WP([7,8 [,A0)=(i%2=0);

A1=WP ([8,10 [, B1)=((i +2)% 2 = 0);

C1 = WP ([7,10 [, C0) = (i +2 <4); C_{1LI} = (1 +2 <4) ≠ False

n=2: B2=WP([7,8[, A1)=((i+2)%2=0);

A2=WP([8,10[,B2) = ((i+2+2)%2=0)

C2=WP([7,10[,C1)=(i+2+2<4); C_{2LI}=(1+2+2<4)=False => Stop

WP (E, j=0) = B_{2LI} = ((1+2)% 2 = 0) = False

2.2. Calcul de la formule: F = True ∧ False = False

2.3. Satisfiabilité: F n'est pas satisfaisable.

3. Conclusion: Il n'existe aucune exécution de Ploop telle que la variable j soit nulle après l'exécution de l'instruction localisée en 7.

- **Exemple 3 : Fractionnement de chemin**

Program *Psplit*

```
1: int x,y,z,t,m;
6: scanf("%d",&x);
7: scanf(''%d'',&y) ;
   if(x<>y)
8: { z=0 ;
     if(x>y)
9:    m=x ;
      else
10:   m=y;
   }else
 { if(x<0)
11: m=-x ;else
12:   m=x;
   }
13: z=1;
   if(m==z)
14:  t=1 ;
   else
15: t=0;
16: m=z+t;
```

VT

Loc	var	Exp
1	X	int_x0
6	X	$x
7	Y	$y
8	Z	0
9	M	X
10	M	Y
11	M	-x
12	m	X
13	Z	1
14	T	1
15	T	0
16	m	z+t

CT

C	LT	LF	LE
x<>y	8	11	13
x>y	9	10	11
x<0	11	12	13
m=z	14	15	16

Figure 3.11. Exemple 3 : Fractionnement de chemin

Vérifions si l'emplacement L = 14 est accessible.

1. Calcul de P0 : P0 = {(True,[1,14 [), (m=z, [14,15[)}

2. Analyse de faisabilité:

2.1. Les plus faibles Préconditions

WP ([1,14[, m=z): WP (13, m=z)=(m=1);

WP(12, m=1) =? Stop car 12 ∈ CT

2.2. Calcul de la formule F: F = True ∧ (m=1)

2.3. Satisfiabilité de F : On ne peut pas décider si F est satisfiable ⇒ P0 doit être étendu.

3. Extension de P0 : P1 = {([1,8[, True); (x<>y, 8,11,13), (m=z, [13,15[)}

4. Analyse de faisabilité: Appelons E1=(x<>y, 8,11,13)

4.1. Les plus faibles Préconditions :

WP (E1, m=1)=(x<>y)∧WP([8,11[,m=1)∨(x=y)∧WP([11,13[,m=1)

WP ([8,11[, m=1): WP(10, m=1): Stop car 10 ∈ (x> y, 9,10,11).

WP ([11,13[, m=1): WP(12, m=1): Stop car 17 ∈ (x <0,16,17,18).

4.2. Calcul de la formule F : F=(m=1)

4.3.Satisfiabilité de F : On ne peut pas décider de la satisfiablité de F.

Donc l'extension de P1 est nécessaire: On étend par (x>y,9,10,11) et (x<0,11,12,13), qui sont opposés ⇒ Split P1 en P2 et P3 :

P2 = {([1,8 [, true); (x<>y, [8,9 [), (x>y, 9,10,11), (m=1, [13,15 [)}

P3 = {([1,8 [, True); (x=y, [11,13[), (x<0,11,12,13), (m=1, [13,15[)}

4.3.1. Exploration de P2: Notons E2=(x>y, 9,10,11); E3=(x<>y, [8,9 [)

- WP (E2, m=1)=(x>y)∧(x=1)∨ (x<=y)∧(y=1)=C1.
- WP (E3, C1)=WP([8,9 [, C1)=C1.
- WP ([1,8 [, C1) = ($x>$y) ∧ ($x=1)∨($x<=$y)∧($y=1)
- WP ([1,8 [, x<>y) = ($x<>$y)

Formule: F2=True∧($x<>$y)∧(($x>$y)∧($x=1)∨($x<=$y)∧($y=1))

Satisfiabilité de F2: F2 est vrai si et seulement si: $x=1 et $y<1 ou $y=1 et $x<=1. Donc F2 est satisfiable.

5. Conclusion: Il existe une exécution permettant d'atteindre la localisation 14. Le processus est stoppé, il n'est pas nécessaire d'explorer le chemin P3.

Le processus suivi dans cet exemple est illustré par la figure 3.12 :

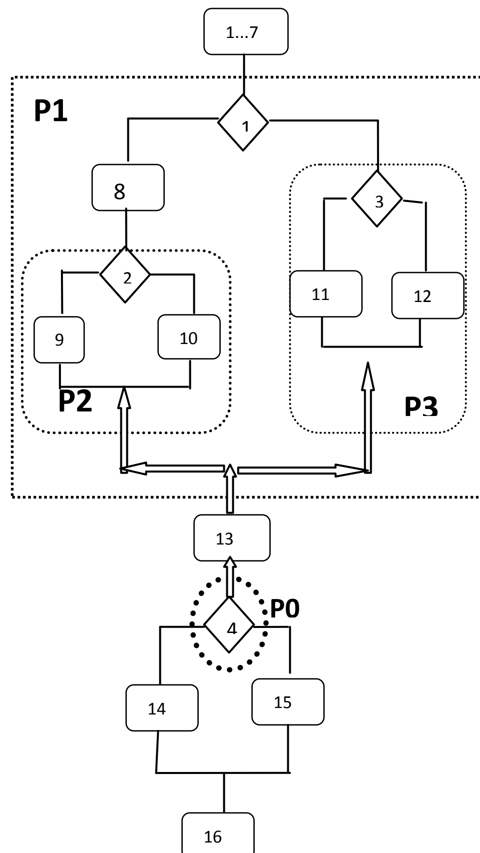


Figure 3.12. Illustration de BIMC sur l'exemple 3.

3.2.5. Procédures et fonctions

Les fonctions sont modélisées de manière indépendante du programme appelant, et de la même manière que les programmes ordinaires. Il existe deux cas concernant la position de la localisation L, exprimant la propriété à vérifier, relativement à une fonction F:

• **La localisation L est dans le corps de la fonction:** Appelons L0 la localisation de l'appel de la fonction dans le programme appelant. Pour atteindre L, L0 doit être atteint en premier. Donc, le problème se transforme en deux parties: d'abord, vérifier l'accessibilité de L0 dans le programme appelant. Si L0 est accessible alors vérifier l'accessibilité de L dans la fonction. La formule F est exprimée avec des paramètres effectifs au lieu des paramètres formels.

• **La localisation L est en dehors de la fonction :** L'idée est d'effectuer des « résumés » de fonction: « function summaries » nous «capturons» l'effet de l'appel de F sur le programme appelant. Ainsi, puisque les conséquences d'un appel de fonction sont les valeurs retournées et les modifications effectuées sur les variables globales, nous traitons ces deux points de la manière suivante:

1. Les variables locales suivantes: #F et #Ret ayant le même type que la fonction f sont ajoutées dans F.
2. Pour chaque variable globale v modifiée dans F, une variable locale #v est déclarée.
3. Chaque instruction return (exp) dans F est modélisée par exp= #Ret. Puisque plusieurs instructions return peuvent exister dans une même fonction, alors #Ret peut avoir plusieurs expressions.
4. À la fin de F, l'instruction if (#F == #Ret) {#F= #Ret} est ajoutée. Cette ligne de code n'a aucun effet sur les variables, mais elle est utilisée à des fins de vérification.
5. Nous vérifions, par le procédé décrit précédemment, si cette instruction est accessible, ce qui implique le calcul des plus faibles préconditions de (#F=#Ret), ce qui donne comme résultat l'expression de #F en fonction de toutes les expressions possibles de #Ret, en d'autres termes, toutes les valeurs calculées par la fonction F.
6. La même idée est réalisée afin de quantifier l'influence de F sur les variables globales: À la fin de F, pour chaque variable globale modifiée dans F, l'instruction if (#v==v) {#v=v} est ajoutée, et nous vérifions si cette localisation est accessible.

7. Dans le processus d'analyse de faisabilité, dans chaque prédicat, l'appel de F est remplacé par la formule obtenue en (5), exprimée avec les paramètres effectifs au lieu des paramètres formels. (voir l'exemple ci-dessous)

Exemple: Considérons une fonction max où d est une variable globale:

---

int max (int x; int y)

```
3: {int m;
// Variables de Vérification
4: int  #max, #Ret,  # j;
   if (x>y)  7: m=x;
      else    8: m=y;
   if  (m>0)  9: d=m+1;
      else   10: d=m-1;
11:  return (m);
// instruction de vérification:
if (#max==#ret)  12: #max=#Ret;
if (#d==d)           13: #d=d;}
```

loc	V	Exp
1	X	int_x0
2	Y	int_y0
3	M	int_m0
7	M	X
8	M	Y
9	D	m+1
10	D	m-1
11	#Ret	M
12	#max	#Ret
13	#d	d

C	LT	LF	LE
x>y	7	8	9
m>0	9	10	11
#max=#Ret	12	-1	13
#d=d	13	-1	14

Figure 3.13. Sous-programmes

Comme nous l'avons expliqué précédemment, nous appliquons l'approche décrite dans les sections précédentes pour vérifier si les localisations L1=12 et L2=13 sont accessibles. En appliquant successivement les plus faibles préconditions, on obtient:

Pour L1: $F1=(x>y)\wedge(\#max=x)\vee(x<=y)\wedge(\#max=y)$. F1 représente exactement toutes les valeurs possibles retournées de la fonction max.

Pour L2: $F2= (x> y)\wedge((x> 0)\wedge(\#d= x +1)\vee(x<= 0)\wedge(\#d=x-1))\vee(x> y)\wedge$
$(((x> 0)\wedge(\#d=x+1)\vee (x <= 0) \wedge (\# d = x-1))$.

F2 résume toutes les modifications effectuées sur la variable #d (représentant la variable globale d). Supposons maintenant que, dans le programme appelant nous avons les deux instructions conditionnelles: if (M1== max (a, b)){...} et if (d> 3){... .}; En remplaçant les paramètres formels par les paramètres effectifs, la condition if(M1 == max (a,b)) devient: if $(a>b)\wedge(M1=a)\vee(a<=b)\wedge(M1=b)$, et la condition if(d> 3) devient :

If $(a>b)\wedge((a>0)\wedge(a+1>3)\vee(a<=0)\wedge(a-1>3))\vee$
$(a<=b)\wedge((b>0)\wedge(b+1>3)\vee(b<=0)\wedge(b-1>3))$

3.2.6. Pointeurs et alias

La manipulation des pointeurs constitue un problème crucial dans la vérification des programmes. Dans notre approche, nous regroupons les références des variables dans des ensembles représentant des classes d'équivalence. Chaque classe a un élément représentant. Toute modification d'un élément de la classe s'exprime sur le représentant de celle-ci signifiant que tous les éléments de la classe considérée sont modifiés de la même manière. Cette méthode nous permet de résoudre le problème des alias de façon très naturelle. Ainsi, lors du calcul de la table VT, nous effectuons les traitements suivants :

1. Pour toute première affectation de &x à une variable v située à une localisation l i.e. l :v=&x , la classe correspondant à &x: C&x = {(v, l)} avec &x son élément représentant est créée.
2. Pour chaque affectation de la forme li: a = b tels que b est un élément d'une classe donnée C, le couple (a, li) est ajouté à la classe C.
3. Pour chaque variable x, chaque affectation ayant la référence de x comme membre droit i.e. ayant la forme li: v=&x, a pour effet d'insérer dans la classe correspondant aux références de x, l'éléments (v,li). ce qui signifie que v est une référence de x depuis la localisation li.
4. Chaque affectation de la forme *c=d telle que c est un élément d'une classe ayant comme représentant l'élément e, a pour effet d'affecter la valeur d à *e. Ainsi, elle est modélisée par: *e= d. De cette manière, toutes les opérations effectuées sur les variables référencées par des noms différents sont exprimées sur le représentant de la classe.

Dans la phase d'analyse de faisabilité, la plus faible précondition d'un prédicat contenant une déréférence est calculée de la même manière que dans les cas précédents, à l'exception que l'élément représentant de la classe d'équivalence est utilisée à la place de la variable.

Exemple

Considérons la portion suivante de programme et générons sa table des variables (sa table de contrôle est générée de la même façon que précédemment) :

Programme

```
1: i = 0
2 :a = &i;
3 :b = &i;
4 :*b = 1;
5 :a = &j;
6 :c = a;
7 :*a = 2;
```

Table des Variables

Loc	var	Exp	Commentaire
1	i	0	-
2	a	&i	Création de C&i insert (a, 2)
3	b	&i	Insertion (b, 3) dans C&i
4	i	1	Car b∈C&i, voir (4)
5	a	&j	Création de C&j insert (a, 5)
6	c	A	Insertion (c, 6) en C&j
7	j	2	Car a∈C&j

Figure 3.14. Pointeurs

D'où, les classes d'équivalence créées sont les suivantes: C&i={(a,2), (b,3)} et C&j = {(a,5), (c,6)}. Ainsi, nous avons dans ces deux ensembles toutes les informations concernant les alias. Par exemple, pour la variable a, nous avons: de la localisation 2 à 4, la variable a est une référence à i, tandis que de l'instruction 5 à la fin, c'est une référence à j. Nous avons aussi, par exemple, l'information que de la localisation 3 à 5 a et b sont des alias et de 6 à la fin a et c sont des alias. Le calcul des plus faibles préconditions est effectué sur l'élément représentant, ce qui nous permet de se ramener aux cas précédents (i.e. sans utilisation de pointeurs). Nous avons par exemple: WP ([1,5[, *a= 1) WP = (4, *& i=1)=WP (4, i=1)=(1=1)=True.

3.2.7. Expérimentation

Ayant une propriété de sûreté et un programme C, BIMC s'assure si le code vérifie la propriété, ou trouve une exécution qui montre comment le code viole la propriété. Nous avons testé BIMC sur différents programmes C. Nous avons obtenu des résultats encourageants. Le nombre d'appels au démonstrateur est acceptable. Nous rapportons ici quelques résultats confirmant l'intérêt de la méthode. Pour chaque programme analysé, nous donnons la longueur P0, pour donner une idée sur le nombre de prédicats suivis, et le nombre de fractionnements et d'extensions, ainsi que le nombre d'appels au prouveur de théorème. Le tableau 1 montre que BIMC donne des résultats encourageants concernant le nombre d'appels au prouveur de théorème. Tous les programmes testés contiennent un haut degré d'imbrication.

Program	Lines	Predicates	P0 Length	Split	Extension	TP-calls	Time(sec)
p1.c	320	20	15	2	5	8	0.041
p2.c	432	23	20	7	10	18	0.144
P3.c	450	28	22	7	22	29	0.191
P4.c	482	30	27	5	11	18	0.342
P5.c	567	36	32	0	31	32	0.640
P6.c	700	40	35	12	15	28	0.845
P7.c	730	45	40	18	23	42	1.685
P8.c	807	62	50	15	9	24	0.780
P9.c	950	65	53	20	13	34	1.723
P10.c	1023	68	56	18	63	82	3.285
P11.c	1508	71	63	9	52	61	1.220

Figure 3.15. Résultats Expérimentaux

3.2.8. Discussion et comparaison avec quelques outils

Dans [GHK+06] sont introduits quelques exemples de programmes sur lesquels sont comparées diverses méthodes bien connues dans le domaine de la vérification, telles que : Yogi, Slam, méthodes basées sur le paradigme CEGAR, Lee Yannakakis (basé sur le calcul du quotient de la relation de bisimulation) , Dart et Blast. Nous incluons à cette liste de comparaisons notre méthode : BIMC.

Programme P1 :

```
{   int a,i,c,err ;
   scanf('%d',&a)
    i=0;c=0; err= 0;
   while (i<1000)
  { c=c+i ;
     i=i+1 ;
  }
    if(a<=0) err=1 ;}
```

BIMC ⟹

Loc	Var	Exp
1	a	Int_a0
2	i	Int_i0
3	c	Int_c0
4	err	Int_err0
5	a	$a
6	I	0
7	c	0
8	err	0
9	c	c+i
10	I	i+1
11	Err	1

C	LT	LF	LE
I<1000	9	-2	11
A<=0	11	-1	12

Figure 3.16. Programme P1

Ce genre de programmes est difficile à vérifier à l'aide de Slam, Blast, et toutes les méthodes basées sur l'approche CEGAR. En effet, les prédicats

(i=0),(i=1)…..(i=999) sont générés un à un (1000 itérations) pour pouvoir trouver un contre-exemple. Dart réussit bien dans ce programme. Yogi exécute 1000 fois la boucle pour trouver le contre-exemple. BIMC étudie l'accessibilité de la localisation 11 en commençant par calculer : P0={(True,[1,11[) ,(a<=0,[11,12[)} ; l'analyse de faisabilité de l'ensemble P0 calcule les plus faibles préconditions : WP(a<=0,[1,11[) : Arrêt puisque 11∈CT. D'où extension de P0.

P1={(True,[1,9[),(i<1000,[9,11[)*,(a<=0,[11,12[)}. Soit E=(i<1000,[9,11[)*

WP(E,a<=0)= ? n=0: C0=i<1000 ;D0=a<=0

n=1 ; D1=WP([9,11[,a<=0)=(a<=0) : D1=D0=> Arrêt

D'où WP(E,a<=0)=(a<=0) ; WP([1,9[,a<=0)=($a<=0)

F=($a<=0) cette formule est satisfiable pour toutes les valeurs négatives ou nulles de a. Donc il existe des exécutions pour laquelle l=11 est accessible.

Programme P2 :

```
{ int y,err ;
  err=0 ;
  while(y>0)
 {y=y-1 ;}
  if(false) err=1 ; }
```

Loc	Var	Exp
1	Y	Int_y0
2	Err	Int_err0
3	Err	0
4	y	y-1
5	err	1

C	LT	LF	LE
y>0	4	-2	5
False	5	-1	6

Figure 3.17. Programme P2

L'analyse avec l'algorithme de Lee-Yannakakis ne termine pas. Yogi et BIMC réussissent très rapidement. La condition sous laquelle la localisation 5 est accessible est fausse donc il n'existe aucune exécution menant à la localisation 5.

Programme P3 :

```
{ int  lock,err ;
 lock = L; err=0 ;
if (C1) {  x1 = x1 + 1; }
else {  x1= x1- 1; }
if (C2) { x2 = x2 + 1;}
else { x2 = x2 - 1;}
if (C3) { x3 = x3 + 1;}
else { x3 = x3- 1;}
…..
if (Cn) { xn = xn + 1;}
else { xn = xn- 1;}
if (lock != L) err=1 ; }
```

Loc	Var	Exp
1	Lock	Int_lock0
2	Err	Int_err0
3	Lock	L
4	Err	0
5	X1	X1+1
6	X1	X1-1
7	X2	X2+1
8	X2	X2-1
9	X3	X3+1
10	X3	X3-1
11	Err	1

C	LT	LF	LE
C1	5	6	7
C2	7	8	9
C3	9	10	11
lock≠L	11	-1	12

Figure 3.18. Programme P3

Yogi et BIMC vérifient ce programme en n itérations, Dart nécessite 2^n cas de test.

Programme P4

```
{ int x,y,err ;
  x=0;y=0;err=0;
  while(y>=0)
  { y=y+x ; }
  if(true) err=1 ;
}
```

BIMC

Loc	Var	Exp
1	X	Int_x0
2	Y	Int_y0
3	Err	Int_err0
4	X	0
5	Y	0
6	Err	0
7	Y	Y+x
8	Err	1

C	LT	LF	LE
y>=0	7	-2	8
True	8	-1	9

Figure 3.19. Programme P4

Yogi ne termine pas sur cet exemple. Il génère les prédicats : (y>=0) ; (y+x>=0) ;(y+2x>=0) ;… BIMC termine immédiatement, mais fournit un résultat qui n'est pas correct. En effet, il ne teste que la condition (true=true) et il déduit que la localisation d'erreur est accessible, alors que le programme P4 s'engage dans une boucle infinie et n'atteigne jamais la localisation 8.

En conclusion de cette section : Nous avons présenté une nouvelle approche pour le problème de la vérification de programmes. Celle-ci présente plusieurs contributions:

1. La méthode de modélisation des programmes, ASMA, est aussi simple qu'avantageuse. En effet, elle permet de manipuler un programme très facilement. On peut caractériser par un ensemble de conditions, chaque région du programme indépendamment du reste du programme et sans être obligé de traverser le programme entièrement depuis le début.

2. La construction progressive de l'ensemble P, en commençant par P0 et en étendant P uniquement en cas de besoin, permet de n'examiner que les parties nécessaires du programme. Cela constitue une sorte d'abstraction, puisque nous écartons toutes les régions du programme qui ne sont pas nécessaires à notre objectif.

3. La procédure d'analyse de faisabilité présente plusieurs avantages:
   - Le calcul de la plus faible précondition permet d'utiliser à chaque localisation, la valeur la plus récente de chaque variable.
   - Un élément clé de la méthode de calcul des plus faibles préconditions successives de tous les éléments de P jusqu'à arriver à la localisation à l'entrée de P. En fait, nous avons un grand avantage dans l'accumulation de la vérification de la satisfiabilité dans un point unique au lieu de vérifier chaque condition séparément, ce qui nécessite un grand nombre d'appels de prouveur de théorème.

4. Notre approche est compositionnelle, ceci constitue une caractéristique très importante. Les fonctions définies dans un programme sont analysées indépendamment du programme appelant en calculant des résumés de fonctions. Ainsi, dans le cas de plusieurs appels à une même fonction, une seule analyse de la fonction est effectuée.

5. Les pointeurs sont représentés et manipulés d'une manière simple et naturelle. Beaucoup d’informations ‘points-to’ et d’alias peuvent être déduites de la table des variables sans effort important.

Dans ce qui suit, nous introduisons une autre approche de vérification de programmes. Celle-ci utilise une méthode évolutionnaire : les algorithmes génétiques, pour rechercher, dans l’espace des exécutions possibles, celle qui faillit à la propriété de sûreté.

3.3. Une approche évolutionnaire pour la vérification de Programmes [ATK11]

3.3.1. Principe et motivations

La recherche d'une exécution vérifiant une propriété dans un très grand ensemble de données possibles nous suggère d'adopter le même raisonnement que celui des méthodes approchées où l'ensemble de recherche est scruté en s'approchant de plus en plus de la solution optimale. Dans ce qui suit, nous utilisons une méthode évolutionnaire pour définir les données d'entrées adéquates pour tenter de générer une exécution vérifiant une propriété spécifiée. Nous utilisons un algorithme génétique pour vérifier les propriétés de sûreté de programmes C. La littérature concernant l'utilisation des méthodes évolutionnaires pour la vérification de programmes est très pauvre. Dans [GK04] et [ACF08] sont présentés des algorithmes génétiques pour la vérification de programmes. Dans les deux travaux les exécutions sont étudiées en termes de chemins. Ainsi, un individu est un chemin d'exécution possible. Nous adoptons une démarche différente. Nous caractérisons une exécution en termes de valeurs d'entrée. Ainsi, tout ensemble de valeurs d'entrée constitue une exécution possible, et par conséquent, un individu possible. Les programmes sont modélisés de la même façon que dans la section précédente. Le problème de la vérification de propriété de sûreté est également exprimé en termes d'accessibilité d'une localisation erronée : L. Habituellement, les méthodes évolutionnaires, définissent un objectif à atteindre et génèrent de façon itérative des solutions qui sont au fur et à mesure améliorées jusqu'à atteindre une solution « optimale ». Nous définissons notre objectif de la façon suivante : nous calculons ACCESS : une chaîne où chaque position représente la valeur requise des gardes des éléments de CT pour atteindre la localisation L, nous l'appellerons «chaîne d'accès" de L. Ensuite, l'algorithme génétique génère itérativement des populations de solutions qui tentent de fournir une exécution qui est "conforme" à ACCESS. Chaque individu est un ensemble des valeurs des variables d'entrée. Nous nous sommes inspirés de l'interprétation abstraite par intervalles, ainsi, pour chaque individu, au lieu de représenter chaque variable d'entrée par une valeur unique, nous représentons chaque entrée par un intervalle. Ceci est justifié par les points suivants :

1. Cette représentation permet une meilleure couverture de l'espace de recherche des solutions ce qui permet une analyse plus exhaustive.
2. La probabilité de trouver la solution est plus élevée. En effet, la solution est d'abord cernée dans un intervalle puis celui-ci est de plus en plus affiné.
3. Cette méthode nous permet de générer à chaque population, une progéniture qui est meilleure que la précédente. En effet, nous avons défini sur les intervalles, des opérateurs de croisement et de mutation garantissant comme résultat des individus meilleurs que leurs antécédents.

3.3.2. Algorithme Génétique pour la Vérification de Programme

3.3.2.1 Algorithmes Génétiques

Les algorithmes génétiques [Hol75] sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de l'évolution naturelle : sélection, croisements, et mutations. Un algorithme génétique recherche le ou les extrema d'une fonction définie sur un espace de données. Cinq éléments sont nécessaires pour travailler avec les algorithmes génétiques :

1. Un principe de codage des individus de la population. Cette étape associe à chacun des points de l'espace d'état une structure de données. Elle se place après une phase de modélisation mathématique du problème traité. La qualité du codage des données a une grande influence sur le succès des algorithmes génétiques. Le codage binaire est le premier codage qui a été défini, il est le plus utilisé.
2. Un mécanisme de génération de la population initiale. Ce mécanisme doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures. Le choix de la population initiale est important car il peut rendre plus ou moins rapide la convergence vers l'optimum global. Dans le cas où l'on ne connait rien du problème à résoudre, il est essentiel que la population initiale soit répartie sur tout le domaine de recherche.
3. Une fonction à optimiser. Celle-ci retourne une valeur appelée fitness ou fonction d'évaluation de l'individu.

4. Des opérateurs permettant de diversifier la population au cours des générations et d'explorer l'espace d'état. L'opérateur de croisement recompose les gènes d'individus existant dans la population, l'opérateur de mutation a pour but de garantir l'exploration de l'espace des états.
5. Des paramètres de dimensionnement : Tels que : la taille de la population, le nombre total de générations, un critère d'arrêt et les probabilités de sélection des individus pour les croisements et les mutations.

Un algorithme génétique commence par générer une population initiale d'individus, et effectue de plus en plus d'amélioration par mutations et croisement jusqu'à atteindre la solution optimale ou dépasser un time out auquel cas la recherche sera interrompue sans trouver de résultat optimal.

3.3.2.2 Algorithme Génétique pour la vérification de programme

Nous utilisons l'algorithme génétique décrit par la figure 3.19. Notre algorithme commence par une population d'individus générés de façon aléatoire. Chaque individu représente une initialisation possible des variables d'entrée. Pour chaque individu i, nous calculons sa chaîne d'accès $Chain_i$ enregistrant la séquence d'éléments de la table CT exécutée par i. L'évaluation de la qualité d'un individu est réalisée par une fonction fitness. Celle-ci mesure la distance entre ACCESS et $Chain_i$. L'objectif est atteint si nous trouvons un individu i* tel que la distance entre $Chain_{i*}$ et ACCESS est égal à zéro. Dans le cas contraire, la population doit être améliorée. Nous définissons un opérateur de combinaison et trois opérateurs de mutation.

3.3.2.2.1 Algorithme

Algorithme 7: Algorithme Génétique pour la vérification de programme

```
1: Inputs: A program Pg, a location L, max_iter, CardPop
2: Outputs: Eventually An execution leading to L.
3: Initializations: Pop0 : a population of individuals
4: Compute The acces chain of L : ACCESS
5: Generate an initial Population ; i=1; success=False;
6: while (i<=max_iter) and (success=False)
7: { j=1;
8:   while(j<=CardPop) and (success=False)
9:   {      Compute Chain[j]
10:          Fitness[j]=Distance (ACCESS ,Chain[j])
11:          If Fitness[j]=0 then success=True
12:                       else j=j+1;
13:   }
14:   If (success=False)
15:         { Recombination();
16:           Narrowing();
17:           StrongMutate();
18:           WeakMutate();
19:           i:=i+1;
20:         }
21: }
22: If  Success
23: The set of input values is an execution  allowing to
        reach L ; Chain[j] is a path to L.
24: Else  No solution found.
```

Figure 3.20. Algorithme Génétique pour la vérification des programmes

3.3.2.2.2 Calcul de la chaîne d'accès de la localisation erronée

Soit L une localisation ayant la chaîne d'accès ACCESS. Celle-ci est une chaîne constituée des caractères '1 ', '0' ou 'x'. Notons l'expression «doit être égale» par «≡». Nous avons:

$$ACCESS\,[i] = \begin{cases} '1' & \text{Si } C[i] \equiv \text{True.} \\ '0' & \text{Si } C[i] \equiv \text{False.} \\ 'x' & \text{Sinon (les deux sont admis)} \end{cases}$$

Exemple : Dans le programme Prog de la figure 3.1, la chaîne d'accès de la localisation : L=14 est : Access=x01. Ceci signifie que dans le premier bloc (z=x,7,9,11) l'une ou l'autre des deux branches pourrait être exécutée de façon indifférente ; dans le deuxième bloc :(z<3,12,13,16), la branche else doit être exécutée; et finalement, la condition (t=-1) doit être vraie pour exécuter la branche then du bloc(t=-1,14,15,16).

3.3.3. Individus

3.3.3.1 Représentation

Pour un programme ayant N variables en entrée, chaque individu de la population sera représenté par un ensemble de N intervalles, chacun d'eux est associé à une variable d'entrée.
Exemple : Pour le programme précédent un individu possible est : x=[0,50[; z=[-20,100[

3.3.3.2 Calcul de la chaîne d'accès d'un individu

Au préalable, il est important d'effectuer les deux remarques suivantes:

1. Pour un individu donné, la valeur des gardes de CT n'est pas toujours connue. Par exemple, pour l'individu i tel que x = [-100,50] et z = [0,10], la valeur de x> z n'est pas connue.

2. I l n'est pas toujours nécessaire de connaître les valeurs de vérité de toutes les conditions de CT. Par exemple si une exécution suit la branche 'else' d'une conditionnelle, tous les blocs qui pourraient exister dans la branche 'Then' ne seront pas évalués.

Notons $Chain_i$ la chaîne d'accès de l'individu i. $Chain_i$ contient les caractères suivants : '1 ', '0', 'u' ou 'x' signifiant respectivement: True, False, inconnu ou non requis. A la rencontre de la première valeur 'u' le calcul de la chaine d'accès est interrompu en remplissant toutes les positions restantes par 'u'. Soit $\omega_i(C[k])$ la valeur de C[k] pour l'individu i. La chaîne d'accès de i notée $Chain_i$ est la chaîne: $a_1a_2a_3$ ..a_n telle que:

$$a_k = \begin{cases} '1' \text{ si } \omega_i(C[k]) = True \\ '0' \text{ si } \omega_i(C[k]) = False \\ 'u' \text{ si } \omega_i(C[k]) \text{ est inconnue ou } \omega_i(C[k-1]) = 'u' \\ 'x' \text{ si } C[k] \text{ n'est pas requis} \end{cases}$$

3.3.3.3 Chemin d'exécution

Notons P_i, le chemin d'exécution de l'individu i. Nous définissons le chemin d'exécution P_i comme la succession des blocs de CT « déclenchés » par les valeurs d'entrée de l'individu i. P_i est défini par l'union: $P_i=P_i[1] \cup P_i[2] .. \cup P_i[m]$, où $P_i[k]$ est la séquence des instructions déclenchée par l'individu i, dans le bloc numéro k de CT. Il est à noter qu'il existe une interdépendance entre le chemin d'exécution et la valeur de la condition d'un bloc de CT. Commençons donc par définir chaque élément de P_i

$$P_i [k] = \begin{cases} Then (k) \text{ si } \omega_i(C[k])= '1' \text{ et } CT[k] \in (IT \cup ITE) \\ Else (k) \text{ si } \omega_i(C[k])= '0' \text{ et } CT[k] \in ITE \\ \varnothing \text{ si } a_k = 'x' \\ (Body (k))^n \text{ si } CT[k] \in LOOP \text{ et } \omega_i(C[k]) = '1'. \end{cases}$$

n est le nombre d'itérations. $(Body (k))^n$ représente l'union de l'intervalle Body(k) n fois.

3.3.3.3.1 Calcul des valeurs des gardes de CT

Nous utilisons les plus faibles préconditions pour effectuer des exécutions symboliques. Cela permet de calculer simplement la garde requise dans le point désiré du programme au lieu de parcourir tout le programme depuis le début jusqu'à l'endroit considéré. Appelons P_{ik} le préfixe de longueur k du chemin P_i. $\omega_i(C[k])$: La valeur de la garde C[k] pour l'individu i, est calculée récursivement de la manière suivante :

1 – Cas où $CT[k] \notin LOOP$:

$$\omega_i(C[k]) = \begin{cases} WP ([1, Begin (k) [) \text{ Si } k = 1 \\ WP (P_i (k-1), C[k]) \text{ Sinon} \end{cases}$$

2 – Cas où CT[k]∈LOOP: Soit CT[k]=(C,LT,-2,LE)*, nous notons $\omega_i(C[k])_j$ la valeur de C[k] dans le chemin P_i à l'itération j:

$$\omega_i(C[k])_j = \begin{cases} \omega_i(C[k]) \text{ Si } j=1 \\ WP(P_i\ (k-1) \cup [LT,LE[j-1,C[k]) \text{ Si } j > 1 \text{ et } \omega_i(C[k])_{j-1}=True \end{cases}$$

Le nombre d'itérations de la boucle est le plus petit entier n tel que $\omega_i(Cd\ [k])_{n+1}$ = False

3.3.3.3.2 Opérations sur les intervalles

Les données manipulées par les individus étant sous forme d’intervalles, nous adoptons les mêmes définitions pour les opérations arithmétiques que dans l'interprétation abstraite par intervalle [CC92]. Les opérations logiques sont nos propres définitions puisque nous utilisons une logique à trois valeurs de vérité : True, False et Unknown.

Opérateurs arithmétiques	**Opérateurs relationnels**	**Table de vérité**				
n= [n, n]	([a,b]=[c,d])=T if(a=b=c=d)	F	G	F∨G	F∧G	¬G
[a, b]+[c, d] = [a + c, b + d]	F if [a,b]∩[c,d]=∅	1	u	1	u	u
[a, b]- [c, d] = [a − d, b − c]	U else	0	u	u	0	u
-[a, b] = [−b,−a]	([a,b]<[c,d])=T if (b<c)	u	u	u	u	u
[a, b]*[c, d] =[Min,Max]	F if (d<a)					
Où :	U else					
Min=min(ac,ad,bc,bd)	([a,b]>[c,d])= F if (b<c)					
Max=max(ac,ad,bc,bd)	T if (a>d)					
	U else					

Figure 3.21. Opérations sur les intervalles

Les opérations : union, intersection, inclusion et appartenance pour les intervalles sont les mêmes que celles définies de façon standard.

3.3.4 Population Initiale

Afin de garantir certaines propriétés souhaitables, la population initiale est générée de telle manière à garantir les points suivants :

1 – Diversité

2 - Qualité acceptable

Chaque individu créé est d'abord évalué afin de vérifier les deux propriétés précédentes. Pour garantir la diversité, nous privilégions les individus ayant des valeurs différentes dans les positions de la chaîne ACCESS représentées par 'x' étant donné que ce sont ces positions qui peuvent fournir des chemins d'accès différents à la même localisation. Un individu a une qualité acceptable si sa chaîne d'accès présente au moins une position conforme à celle de ACCESS. Il est également avantageux d'utiliser des intervalles de grande taille dans la phase d'initialisation, et de les rétrécir progressivement. Ainsi, soit |Pop| le nombre d'individus de la population initiale. Donc, dans la phase d'initialisation, nous générons | Pop |+K (K> 0) individus, mais nous ne conservons que les |Pop| "meilleurs".

3.3.5. Evaluation des individus

Pour un individu i, la fonction de fitness, Fitness(i) mesure la distance entre ACCESS et Chaini. Le calcul de la valeur de Fitness est effectué de la manière suivante: Soit Fit une chaîne telle que:

$$Fit[k] = \begin{cases} 0 & \text{Si (ACCESS[k]='x') Ou (ACCESS[k]=} Chain_i \text{[k])} \\ 1 & \text{Sinon} \end{cases}$$

Fitness (i) est le nombre décimal obtenu en convertissant le nombre binaire représenté par Fit.

Exemple : Soit ACCESS = 10xx1, et soit i un individu tel que chaini=1101u; Fit = 01001; Fitness (i) = 9.

On remarque que la fonction de Fitness représente fidèlement la distance entre le comportement souhaité et le comportement de l'individu considéré. En effet, si on considère par exemple deux individus i1 et i2 tels que FIT1=1000 et FIT2= 0001 en dépit du fait que ces deux individus ont tous deux une position en défaut : qui ne correspond pas avec ACCESS, leur valeurs de Fitness doivent être différentes parce que le premier individu a 'échoué' dans la première garde, donc il a pris un chemin complètement différent de ACCESS, et il représente une exécution qui est complètement divergente. Alors que i2 a égalé avec ACCESS jusqu'à la dernière position de sorte qu'il est plus proche de ACCESS. Donc, i2 est meilleur que i1, ce qui est effectivement exprimé par la fonction de Fitness puisque: Fitness (i1) = 8 et Fitness (i2) = 1. Rappelons que le but est de trouver un individu i* tel que Fitness (i*) = 0.

3.3.6. Amélioration de la population

Pour améliorer la population, nous adoptons une approche guidée qui augmente la probabilité des individus obtenus pour être effectivement meilleurs que leurs antécédents. Toutefois, étant donné que le croisement et la mutation peuvent être effectués plusieurs fois dans un algorithme génétique, ils doivent donc être aussi simples que possible. Donc, nous procédons à une amélioration graduelle. Elle consiste à «corriger» la première position non conforme à ACCESS de chaque individu de la population : nous appellerons une telle position 'position défectueuse' : Une position dont la valeur dans la chaîne d'accès de l'individu et dans ACCESS sont différentes, et sa valeur dans ACCESS est '0 'ou '1'. Une position défectueuse pourrait être une position erronée ou inconnue. Nous classons les individus en : 'Fit' ou 'Unfit' : selon qu'ils soient bons ou mauvais ; et nous classons leurs positions défectueuses en 'Erroneous' ou 'Unknown' : selon que ces positions défectueuses soient erronées ou inconnues. Notons FU, FE, UU, UE respectivement les catégories obtenues des individus: Fit avec des positions Unknown, Fit avec des positions Erroneous, Unfit ayant des positions Unknown et enfin, Unfit ayant des positions Erroneous.

3.3.6.1 Croisement

L'opérateur de croisement est appliqué à la classe FU pour corriger progressivement les positions défectueuses d'un individu. Soient i1 et i2 deux individus tels que i1 a la position p comme première position inconnue et p n'est pas une position défectueuse pour i2. L'idée est d'utiliser les données de l'individu i2 pour corriger la position p inconnue de i1. Toutefois, étant donné que dans un programme les variables sont fortement corrélées les unes aux autres, donc, modifier certaines données d'un individu pour corriger une garde tout en altérant négativement les autres gardes. Pour éviter cette situation, nous modifions les données correspondant à une position défectueuse de façon conservative. Donc, nous effectuons une intersection entre les données, apparaissant dans la position p, de i2 et celles de i1. Par conséquent, toutes les gardes qui avaient une valeur connue conservent leur valeur, et celles qui avaient des valeurs inconnues pourraient avoir des valeurs connues. Par conséquent, l'opérateur de croisement est défini comme suit: Soient les individus: i1, i2 et i3 tels que p est une position défectueuse de i1, et soit x_{1i}, x_{2i}, x_{3i} les intervalles des

valeurs de la variable d'entrée x_i respectivement pour les individus i_1, i_2 et i_3.

Croisement (i_1, i_2, p) = (i_3, i_2) telle que pour toutes les variables d'entrées x_i:

$$x_{3i} = \begin{cases} x_{1i} \cap x_{2i} & \text{Si } x_i \text{ apparaît dans C[p]} \\ x_{1i} & \text{sinon.} \end{cases}$$

Exemple: Soit ACCES=1x1101; soit i_1 un individu tel que $Chaini_1$= 011001, les positions défectueuses de i_1 sont: 1 et 4. Le point de croisement est donc la position 1. Soit i_2 un individu tel que $Chaini_2$ = 101000, 1 n'est pas une position défectueuse de i_2. Donc, nous utilisons i_2 pour corriger la position 1 de i_1. Les variables d'entrée qui apparaissent dans C[1] sont y et z, donc:
croisement (i_1, i_2, 1) = (i_3, i_2) tel que: $y_3 = y_1 \cap y_2$ et $z_3 = z_1 \cap,z_2$ où y_i et z_i sont les intervalles de variables y et z de l'individu i.

3.3.6.2 Mutation

Nous définissons trois opérateurs de mutation: la mutation faible, la mutation forte et le rétrécissement :

- **La mutation faible :** Elle effectue une petite perturbation sur les individus de la classe FE. Elle consiste à modifier l'intervalle d'une variable unique. La variable candidate au changement doit apparaître dans la position défectueuse.

- **La mutation forte** : Elle est utilisée pour améliorer la catégorie UE. Elle consiste à changer aléatoirement toutes les valeurs des variables. Ce qui correspond à générer un nouvel individu.

- **L'opérateur de rétrécissement :** Cet opérateur consiste à réduire, de diverses manières, les valeurs d'entrée des intervalles. Il est utilisé pour éliminer des positions inconnues des individus de la catégorie UU, car les valeurs inconnues sont dues aux grandes tailles des intervalles.

3.3.7. Expérimentation

Nous avons effectué plusieurs expérimentations pour évaluer notre approche. Les paramètres de l'algorithme génétique, comme le nombre maximal d'itérations, la taille de la population, et la proportion de sélection des individus, sont ajustés au cours de l'expérimentation. Nous rapportons sur la figure 3.22 les résultats obtenus pour 5 programmes où la propriété de sûreté n'est pas vérifiée (Unsafe), et dans chaque programme Prn, la chaîne d'accès ACCESS de la localisation erronée a une longueur n. La colonne Prouvé indique si notre outil a trouvé le programme unsafe (Y) ou non (N). Nous rapportons le nombre d'itérations, la taille de la population et le temps d'exécution.

Programme	Prouvé	Taille Population	Nombre d'Iterations	Temps d'exécution (sec)
Pr5	Y	50	3	0.05
Pr8	Y	50	3	0.06
Pr10	Y	100	4	0.1
Pr16	Y	100	4	0.06
Pr20	Y	100	200	5

Figure 3.22. Résultats Expérimentaux. Algorithme génétique pour la vérification de programmes :

3.3.8. Discussion et comparaison avec d'autres méthodes

Dans cette section, nous avons présenté une méthode évolutionnaire pour la vérification de programme. Dans [GK04] et [ACF08] sont également présentées deux méthodes évolutionnaires pour la vérification de programmes. Ces travaux considèrent les chemins d'exécution comme individus. Nous pensons qu'il est plus naturel de considérer les valeurs des entrées comme individus, vu que ce sont ces dernières qui engendrent l'un ou l'autre des chemins d'exécution. L'utilisation des intervalles permet une couverture très efficace de l'ensemble des exécutions possibles. Toutefois, comme il est également le cas pour l'interprétation abstraite par intervalle, ceci restreint la méthode à une classe de fonction. Par exemple, nous ne pouvons pas utiliser les fonctions trigonométriques. Ce qui constitue une limitation de la méthode.

Chapitre 4

Test des Programmes

4.1 Préliminaires

Le test de logiciel est aujourd'hui, la méthode la plus utilisée pour vérifier la correction des logiciels. Selon l'IEEE « Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus ». Glendford. J. Myers le définit de la façon suivante : « Tester c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts » [Mey79]. Le test est la technique la plus couramment utilisée pour valider les logiciels. Ceci est justifié par le fait, qu'il permet un bon passage à l'échelle, et n'a généralement pas besoin d'une infrastructure importante. Cependant, pour les systèmes complexes cet avantage se retrouve rapidement atténué par le fait qu'il est difficile de développer les «bons» cas de test qui fournissent suffisamment de couverture. Cette difficulté réduit souvent la couverture malgré le recours à de nombreux cas de test. La conséquence en est que le test devient un processus laborieux qui représente généralement environ la moitié du coût total de développement de logiciels [BVZ+11]. Par conséquent, automatiser la génération des cas de test permet non seulement de réduire le coût du test de logiciels, mais augmente aussi sa fiabilité. La génération des cas de test s'apprête bien à l'automatisation et a fait l'objet d'un intérêt particulier de beaucoup de recherches. Elle a également été adoptée dans l'industrie.

4.1.1. Génération automatique de cas de test

Automatiser la génération des cas de test consiste à définir les méthodes et outils permettant de :

1. Générer les données de test : Une donnée de test est un ensemble possible des valeurs en entrée. Une suite de test regroupe l'ensemble des données de test générées. Le problème le plus crucial de cette phase est la pertinence et la non redondance des données de test générées.

2. Exécuter les données de tests et récupérer les résultats pour chaque donnée de test. Chaque couple formé par la donnée de test et le résultat correspondant est appelé cas de test.

3. Utiliser l'oracle de test pour vérifier si le résultat obtenu par un test est correct. Cette étape est très délicate vue qu'il n'est pas toujours aisé d'avoir un moyen pour générer le « vrai » résultat de l'exécution considérée.

4. Vérifier si le critère de couverture souhaité est atteint : Le critère de couverture spécifie le but de la génération automatique des cas de test. Il consiste à couvrir des régions spécifiques du programme. Plusieurs critères de couverture ont été définis :
   - Le critère de couverture de toutes les branches : La couverture de toutes les branches garantit que pour chaque condition du programme, il existe une donnée de test permettant d'exécuter sa branche 'THEN' et une donnée exécutant sa branche 'ELSE'.
   - Le critère de couverture de tous les chemins : La couverture de tous les chemins garantit qu'il existe des données de test pour tout chemin possible du programme. Dans le cas de boucle chaque itération engendre un chemin possible.
   - Le critère de couverture des k-chemins : Comme le nombre de chemins exécutables d'un programme peut être infini, le critère de couverture de tous les chemins est souvent relaxé avec le critère de couverture des k-chemins où seuls les chemins pour lesquels chacune des boucles est dépliée moins de k fois sont à couvrir.

4.1.2. Catégories de Tests

Une multitude de classifications existe pour les catégories de tests. Celles-ci dépendent généralement du niveau d'observabilité du système à tester et de la couverture souhaitée, mais aussi d'autres paramètres tels que la phase du développement du logiciel à laquelle est appliqué le test. Selon les deux premiers critères cités, nous exposons deux catégories principales : le test fonctionnel et le test structurel.

4.1.2.1 Test Fonctionnel

Le test fonctionnel, également appelé test boîte noire, suppose un point de vue externe au système sous test (SUT). La structure du programme est inconnue. Dans l'approche boîte noire, les cas de test sont générés sur la base d'une spécification des entrées. Les données concrètes de test d'entrée sont utilisées afin d'obtenir des résultats du SUT, qui sont ensuite comparés aux résultats attendus. Habituellement, l'oracle, utilise les propriétés fonctionnelles, en comparant les données de sortie avec les valeurs attendues. L'approche est facile en général, mais devient rapidement problématique pour des SUT complexes, surtout si l'ensemble des données pertinentes possibles est grand, et les résultats sont complexes et nécessitent une analyse coûteuse.

Exemple de test fonctionnel : JUNIT[BG07]

L'exemple le plus remarquable des outils de test fonctionnel est JUnit. JUnit a été initialement conçu pour le langage Java, mais par la suite, il a donné naissance à un grand nombre de mises en œuvre pour d'autres langages (cppunit pour C++, PyUnit pour Python, etc.)
L'objectif principal de JUnit est de fournir une infrastructure pour la création efficace et l'exécution des tests de régression. Cet objectif est atteint par la création de classes JUnit de cas de test qui appellent les fonctions à tester et utilisent la librairie JUnit pour mettre en œuvre la propriété. JUnit simplifie l'exécution, le stockage, et la réutilisation des tests. Il offre des primitives pour créer un test à l'aide des assertions et pour gérer des suites de tests. JUnit souffre cependant de quelques problèmes inhérents à ce genre d'outils dont : maîtrise de la taille du programme, problèmes de la sélection des tests, et données de test redondantes ou non pertinentes. Par conséquent, l'efficacité du test dépend crucialement de la qualité des données de test.

4.1.2.2 Test Structurel

Le test structurel s'appuie sur l'analyse du code source du programme. C'est pour cette raison qu'il est désigné par : test boite blanche, puisqu'il suppose connue la structure interne du programme. Le test exhaustif qui consiste à soumettre à un programme toutes les données d'entrée possibles est généralement impraticable car il y en a un très grand nombre, voire une

infinité. Des critères de test ont donc été définis pour orienter la sélection des données d'entrée, en décrivant les propriétés que l'ensemble de données sélectionnées doit satisfaire. Les critères de test permettent de caractériser à posteriori la qualité d'un ensemble de données de test, mais aussi de guider la génération de ces données. Ceux-ci dépendent de la représentation utilisée pour le programme à tester.
Dans notre travail, nous nous intéressons particulièrement aux méthodes structurelles. Nous commencerons donc par exposer quelques méthodes de tests structurels, pour introduire, ensuite, notre méthode de génération automatique de cas de test dans un nuage d'ordinateurs. En effet, une problématique cruciale de la génération automatique des cas de test est l'explosion des chemins d'exécution ce qui rend la distribution de cette tâche sur un ensemble de nœuds, un choix tout à fait fondé.

4.2. Etat de l'art : Méthodes structurelles de génération automatique des cas de tests

Diverses méthodes ont été développées pour rendre la génération des cas de test de plus en plus automatique et performante. Les premières méthodes automatiques de génération de cas de test étaient aléatoires. Celles-ci souffraient de diverses limitations dont : la redondance des cas générés et la non pertinence de ceux-ci relativement à la couverture souhaitée. Par conséquent, de nouvelles approches ont été définies pour palier à ces limitations. Nous pouvons classer les méthodes structurelles de tests en deux grandes familles :

1. Méthodes orientées recherche [GK04] : Elles tendent à explorer l'espace des entrées possibles jusqu``a en trouver une satisfaisante. La recherche se base sur des heuristiques et méta-heuristiques issues de l'optimisation combinatoire pour guider la recherche vers les données de test pertinentes. Ainsi, ont été utilisées les méthodes : recuit simulé, colonies de fourmis, algorithmes génétiques..etc. pour la génération automatique de cas de test.
2. Méthodes orientées contraintes [BCG+09, BH09] : Elles consistent en la traduction de certaines parties du programme en formules logiques dont les solutions sont des données de test pertinentes. La génération automatique par exécution symbolique est l'exemple le plus utilisé de cette famille de test.

4.2.1. Génération automatique de test basée sur l'exécution symbolique

La génération automatique des cas de test par exécution symbolique opère sur le graphe de flot de contrôle du programme à analyser. Le principe général de cette méthode est le suivant :

1. Considérer un chemin du graphe de flot de contrôle.
2. Calculer son prédicat de chemin.
3. Résoudre le prédicat : une solution représente une donnée de test exerçant le chemin.
4. Si la couverture est incomplète, goto 1.

La couverture complète étant généralement impossible, les contraintes suivantes sont habituellement ajoutées au processus de l'exécution symbolique :

1. Borne sur la longueur des chemins
2. Time out sur le solveur de contraintes
3. Gestion de couverture (instructions, branches)

Plusieurs outils basés sur ce principe ont été développés. Dans [GBW06], une méthode de génération de tests pour les langages C et C++ est proposée. Elle se fixe comme objectif de test un élément du code source du programme sous test (instruction, branche) et essaie de générer une donnée de test pour couvrir cet élément. Pour couvrir une instruction, cette méthode ne fait pas le choix à priori d'un chemin particulier. Elle gère les programmes avec des pointeurs [GDB05], des allocations dynamiques [CBG09] et des nombres flottants [BGM06]. La méthode est implantée dans un outil nommé InKa, dont le fonctionnement est le suivant : Le programme à tester est d'abord traduit dans un langage intermédiaire où les instructions complexes sont représentées à l'aide d'instructions plus simples grâce à l'introduction de variables temporaires. Ensuite, ces instructions du langage intermédiaire sont interprétées pour générer un système de contraintes. L'obligation d'atteindre l'instruction souhaitée est également traduite sous forme de contraintes ajoutées au système. La résolution du système de contraintes est lancée. Si InKa détermine que le système de contraintes a des solutions, il retourne une donnée de test permettant d'atteindre l'instruction sélectionnée. S'il détermine que le système de contraintes n'a pas de solution, l'instruction à atteindre est inaccessible (code mort). Si la résolution du système de contraintes excède un temps fixé, la résolution du

système s'arrête sans que l'outil ne puisse ni fournir une donnée de test, ni affirmer que l'instruction du programme à atteindre est inaccessible. Toutes les instructions sont traitées comme des relations entre deux états mémoires. Un état mémoire est une représentation abstraite de la mémoire physique en un point de l'exécution. Les contraintes générées par interprétation d'une instruction traduisent les liens entre la mémoire avant l'instruction et la mémoire après l'instruction, sous forme d'une relation entre ces états. Chaque instruction de contrôle if-then-else est traitée en calculant Cthen et Celse, deux formes conjonctives de contraintes obtenues en interprétant respectivement les instructions dans la partie Then et else. La branche à exécuter peut être imposée par l'objectif. Dans le cas contraire, où les deux branches sont possibles, il est nécessaire de prouver que ($c \wedge Cthen$) ou ($\neg c \wedge Celse$) est inconsistant avec le système de contraintes, imposant ainsi l'un des deux chemins et réfutant le second. Si aucune branche n'est écartée, la mémoire en entrée (respectivement en sortie) de la structure de contrôle est une union disjonctive des états mémoires en entrée (respectivement en sortie) de chacune des branches. Il est clair que le nombre et la taille des états ainsi que la taille des contraintes augmentent de façon exponentielle avec la taille du programme et le nombre d'instruction conditionnelle qu'il contient. Cet inconvénient n'est pas propre à cette approche mais à toutes les méthodes de génération automatique de test basées sur l'exécution symbolique. En effet, bien que les techniques symboliques ont été prouvées être très adéquates pour la génération automatique de cas de test, elles ne parviennent pas à passer à l'échelle de grands programmes [God07]. Ceci est due au fait que le nombre de chemins d'exécution possibles à prendre en considération est trop grand. Or, l'explosion se produit principalement en raison des boucles et les conditions. Ainsi, il est important de mettre au point des stratégies de recherche qui permettent d'atteindre rapidement une couverture de branche du programme. Le défi le plus significatif à relever par le test basé sur les chemins est de savoir comment gérer ce nombre exponentiel de chemins dans le programme. Pour pallier à ces limitations, une nouvelle famille de méthode de génération de test a vu le jour. Celle-ci est basée sur l'exécution symbolique, mais réalisée en parallèle avec l'exécution réelle du programme. Ce genre d'amalgames entre exécutions a été baptisé : exécution concolique.

4.2.2. Génération automatique de cas de test basée sur l'exécution concolique

L'exécution concolique combine exécution CONCrète du programme et exécution symbOLIQUE afin d'explorer les chemins du programme. Récemment, les méthodes concoliques de test [MA05, SA06b] ont été proposées comme une variante d'exécution symbolique où l'exécution symbolique est réalisée simultanément avec l'exécution concrète du programme. Précisément, le programme est exécuté simultanément sur des valeurs concrètes et symboliques, et les contraintes symboliques générées le long du chemin sont simplifiées en utilisant les valeurs concrètes correspondantes. L'exécution concrète dans ce cadre sert à guider l'exploration des chemins du graphe de flot de contrôle, et parfois à aider l'exécution symbolique en approximant la valeur d'expressions complexes telles que les expressions non linéaires. Les techniques concoliques assurent une bonne couverture des comportements d'un programme parce qu'elles tentent de générer une entrée de test unique pour chaque chemin d'exécution possible, évitant, ainsi, la redondance des données de test. Divers outils de test ont été mis en œuvre pour réaliser l'idée de base de l'exécution concolique. Les outils existants peuvent être classés en fonction de l'approche qu'ils utilisent pour extraire des formules symboliques à partir des chemins d'exécution concrète. La première approche pour extraire des formules symboliques de chemin est d'utiliser des machines virtuelles sur lesquelles s'exécutent les programmes cibles. PEX [TdH08], KLEE [CDE08] et jFuzz [JHG+09] sont des outils qui utilisent cette approche. La seconde approche est d'instrumenter le code source cible pour insérer des sondes qui extraient des formules à partir des exécutions concrètes au moment de l'exécution. Les outils qui utilisent cette approche comprennent CUTE[MA05], DART [GKS05], jCUTE [SA06b] et SCORE [KK11]. Différentes méthodes exploitant l'exécution concolique pour la génération de données de test ont été définies, pour tester des programmes écrits en langage C [WMMR05, GKS05, God07, SMA05], des programmes en bytecode Java [SA06], des programmes dans le langage intermédiaire du .NET [TdH08, TS06] et des exécutables [BH08].

4.2.3. Approche distribuée pour la génération automatique de test

Parallèlement à l'amélioration des performances des méthodes de test, s'est accru la complexité des logiciels intensifiant par la même le besoin de développer des logiciels fiables. Cette complexité est telle qu'il est rapidement devenu nécessaire d'utiliser plusieurs calculateurs pour atteindre un degré de couverture acceptable. Un grand nombre de recherches sur l'emploi des plates-formes distribuées pour améliorer le passage à l'échelle des techniques de test concolique ont été proposées. L'idée principale de ces méthodes consiste à répartir les chemins d'exécutions possibles, exprimés par le graphe de flot de contrôle, sur différents nœuds. Dans [KKR11] une technique de partitionnement statique de l'arbre des exécutions est proposée : la répartition des sous arbres sur les nœuds est réalisée une fois pour toutes. Une limitation de l'approche proposée est que les arbres d'exécution partitionnés résultants ne sont pas bien équilibrés. Ainsi, certains nœuds peuvent finir d'explorer les chemins d'exécution symboliques rapidement et devenir inactifs pendant que les autres nœuds prennent de longues périodes pour l'exploration complète. Ceci dégrade les performances globales. En revanche, King [KKR11] et ParSym [SK10] utilisent le partitionnement dynamique des exécutions des programmes. King remplit une file d'attente des sous-arbres des exécutions symboliques, mais l'accélération résultante diminue lorsque le nombre de nœuds augmente au-delà de six [SK10]. ParSym utilise un serveur central qui recueille des cas de test générés à partir des nœuds et distribue les cas de test sur les autres nœuds dont les files d'attente sont vides. ParSym montre une accélération relativement à d'autres outils mais ne parvient pas à l'accélération linéaire. Cloud9 [CZB+09] est un service de test basé sur des techniques d'exécution parallèles concoliques mises en œuvre sur KLEE [CDE08]. Cloud9 utilise le partitionnement dynamique afin d'assurer que les longueurs des files d'attente des travaux de tous les nœuds restent dans une plage donnée. SCORE [KK11] distribue dynamiquement à la demande les cas de test entre plusieurs nœuds.

Notre contribution s'inscrit dans ce cadre. Elle porte sur la parallélisation et la distribution du processus de génération automatique de cas de test.

4.3. Nouvelle approche parallèle de génération automatique de cas de test [AK12b]

Pour faire face au phénomène d'explosion de chemins, il est logique de penser à paralléliser et à distribuer le processus d'exploration des chemins. Nous avons développé une méthode d'exécution parallèle. Celle-ci permettra de diviser la tâche d'exploration des différents chemins d'exécution et d'entreprendre des investigations parallèles.

4.3.1. Exécution symbolique arrière pour la génération automatique de cas de tests

4.3.1.1 Modélisation des chemins d'exécution

Soit Path[i] une chaîne exprimant la valeur des gardes requises de tous les éléments de CT, pour atteindre l'élément CT [i]. Path[i,k] représente le caractère de la position k dans Path[i]. Notons, comme précédemment, l'expression «doit être égale» par «≡». Ainsi, nous définissons Path[i, k] par:

Path[i,k] =
- '1' si C[k] ≡ vrai
- '0' si C[k] ≡ Faux
- 'x' si C [k] ≡ True ou False. Les deux sont possibles
- '*' si la valeur de C [k] n'est pas requise.

Où C[k] est la valeur de vérité de la condition du bloc CT[k]. Chaque caractère 'x' génère deux chemins possibles. Par exemple x00 représente les deux chemins possibles 100 et 000. Donc, si P est une chaîne de chemin contenant nx caractère 'x', alors le nombre de chemins représentés par P est 2^{nx}.

L'utilisation du caractère "*" est nécessaire dans la situation où nous suivons un chemin dans une branche « Else ». Dans ce cas, toutes les conditions qui sont dans la branche « Then » ne sont pas nécessaires et ne doivent pas être évaluées.

Définition (chemin Terminal): Soit CT une table de contrôle comportant n éléments. Un chemin, représenté par une chaîne de chemin Path[i], est dit terminal s'il n'existe pas $j \in [1,n]$ tel que $j \neq i$ et Path[i] est un préfixe de Path[j]. La définition habituelle du préfixe de chaîne est étendue pour

prendre en compte le caractère spécial ‘x’ de manière qu'il puisse être assimilé à la fois aux caractères ’0’ 'et '1'.
Terminal [i] est un prédicat qui est vrai si et seulement si Path[i] est un chemin terminal. Un chemin terminal représente une exécution complète du programme (du début du programme jusqu’à sa fin). Pour chaque chemin tel que Terminal [i] est vrai, nous concaténons ‘x’ à la fin de Path[i] pour indiquer les deux exécutions possibles dans le bloc CT[i].
L'algorithme utilisant une analyse arrière pour générer des cas de test est présenté sur la fogire 4.1.

4.3.1.2 Algorithme d’exécution symbolique arrière pour le Test

Algorithme: BSE-Testing

1: Input: Prog: A Program
2: Output: A set of test cases
3: Initializations: Cases=∅; (VT,CT)=Model(Prog)
4: For i=1 to CardCT
5: Do If (Terminal[i])
6: Then Compute (Path[i])
7: For each path P_{ir} of Path[i]
8: Do CP_{ir}=True;
9: For k=1 to i-1
10: Do $CP_{ir}=CP_{ir} \wedge WP(Path_k, path[i,k])$
11: Resolve(CP_{ir}, Cas_{ir});
12: $Cases=Cases \cup Cas_{ir}$;
13: End.

Figure 4.1. Algorithme BSE-Testing

Cet algorithme exécute les étapes suivantes :

1. Pour chaque élément CT[i] de CT calculer la chaîne de chemin Path[i].
2. Pour chaque chemin P_{ir} engendré par Path[i], calculer la plus faible précondition de P_{ir} le long du chemin exprimé par Pathr[i].
3. Déduire les valeurs d'entrée requises pour que l’exécution suive le chemin P_{ir}.

L'exemple présenté par la figure 4.2. illustre la méthode :

```
1: scanf("%d%d",&z,&x)
   if (z==x)
3:   { t=x+1;
4:     y=x-1; }
     else
5:   { t=x-1;
6:     y=x+1 ;}
7:  y=t+y;
    if (z>=3)
8:   { t=z ; }
    else
9:   { t=-z;
      if (t=-1)
10:       {y=0;}
      else
11:       { y=1;}
     }
```

Loc	var	exp
1	z	\$z
2	x	\$x
3	t	x+1
4	y	x-1
5	t	x-1
6	y	x+1
7	y	t+y
8	t	Z
9	t	-z
10	y	0
11	y	1

I	C	CT	CF	LE	Path[i]	Terminal
1	z==x	3	5	7	-	N
2	z>=3	8	9	12	X	N
3	t=-1	10	11	12	x0 : x0x	Y

Figure 4.2. Exemple illustrant BSE

- Path[1] désigné par "-", représente la chaîne vide. En effet, le bloc CT[1] ne nécessite aucune contrainte pour être atteint.
- Path[3] = "x0" signifie que pour accéder à CT[3], la condition (z==x) peut prendre n'importe quelle valeur, alors que la condition (z>=3) doit être fausse. Path[3,1]=C[1]=(z=x)≡True ou False et Path[3,2]=C[2]=(z>=3)≡ False.
- Terminal [3] a la valeur True ; Nous avons donc concaténer 'x' à la fin pour représenter les deux alternatives possibles à partir de CT [3]. Les chemins représentés par Path[3] i.e. 'xox', sont Path1 [3] = "000"; Path2[3] = "001"; Path3[3] = "100" et Path4[3] = "101".

En guise d'exemple, générons un cas de test pour Path1[3] en effectuant une analyse arrière: $WP([1,3[\cup[5,8[,z<3)\wedge WP([1,3[,z\neq x)\wedge WP([9,10[,t\neq -1)= (\$z<3)\wedge(\$z\neq\$x)\wedge(\$z\neq -1)$

Cette formule est satisfiable. En effet, les valeurs d'entrées des variables x et z qui satisfont cette formule sont par exemple, $z=2 et $x=1 ; permettent une exécution suivant le chemin Path1[3] i.e. '000'. Cette exécution suivra donc les branches 'else' des trois instructions conditionnelles du programme. Le même raisonnement est effectué pour les autres chemins. Le nombre de chemins d'exécution du programme étant généralement trop grand, nous procédons à la parallélisation du processus de génération des cas de test et à sa répartition sur plusieurs unités de calcul.

4.3.2. Exécution symbolique arrière parallèle pour la génération de cas de test

Dans ce qui suit, nous présentons un algorithme parallèle pour une exécution symbolique arrière. Un programme est divisé en plusieurs parties assignées à différentes unités de calcul d'un nuage d'ordinateurs. Un nœud particulier : le coordinateur attribue les tâches aux autres nœuds et recueille les résultats finaux. Nous développons un cadre d'exécution distribuée symbolique. Les techniques d'exécution symboliques représentent généralement les exécutions, par des arbres. Le processus de répartition appliqué sur les arbres possède l'inconvénient de nécessiter un partitionnement dynamique pour être bien équilibré [CZB+09]. Notre méthode de représentation des programmes permet une parallélisation plus facile du processus d'exécution symbolique arrière, et garantit un bon équilibre entre les nœuds sans nécessiter un partitionnement dynamique. La génération automatique parallèle des cas de test est attribuée à plusieurs nœuds. Un nœud particulier appelé le coordinateur initie et termine la génération automatique des tests. L'algorithme de la figure 4.4 décrit le coordinateur; nbWorker est le nombre de nœuds. Notre approche vise à atteindre les objectifs suivants :

1. Effectuer un partitionnement statique et bien équilibré.
2. Réduire la redondance dans les calculs.
3. Réduire les dépendances entre les nœuds.
4. Réduire les communications entre le coordinateur et les autres nœuds.

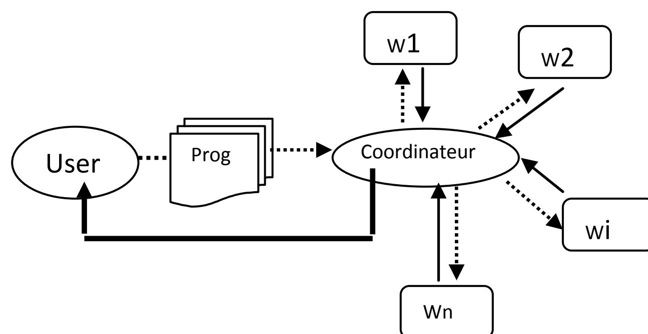


Figure 4.3. Architecture de BSE dans le Nuage d'ordinateurs

Le rôle du coordinateur consiste à :
1. Modéliser le programme par la paire (VT, CT),
2. Calculer les chaînes de chemin: Path[i] pour chaque élément de CT.
3. Répartir les tâches d'une manière bien équilibrée entre les nœuds.

La tâche de chaque nœud consiste à calculer les cas de test des chemins qui lui sont confiés et d'envoyer les résultats au coordinateur.

Algorithme : Coordinateur

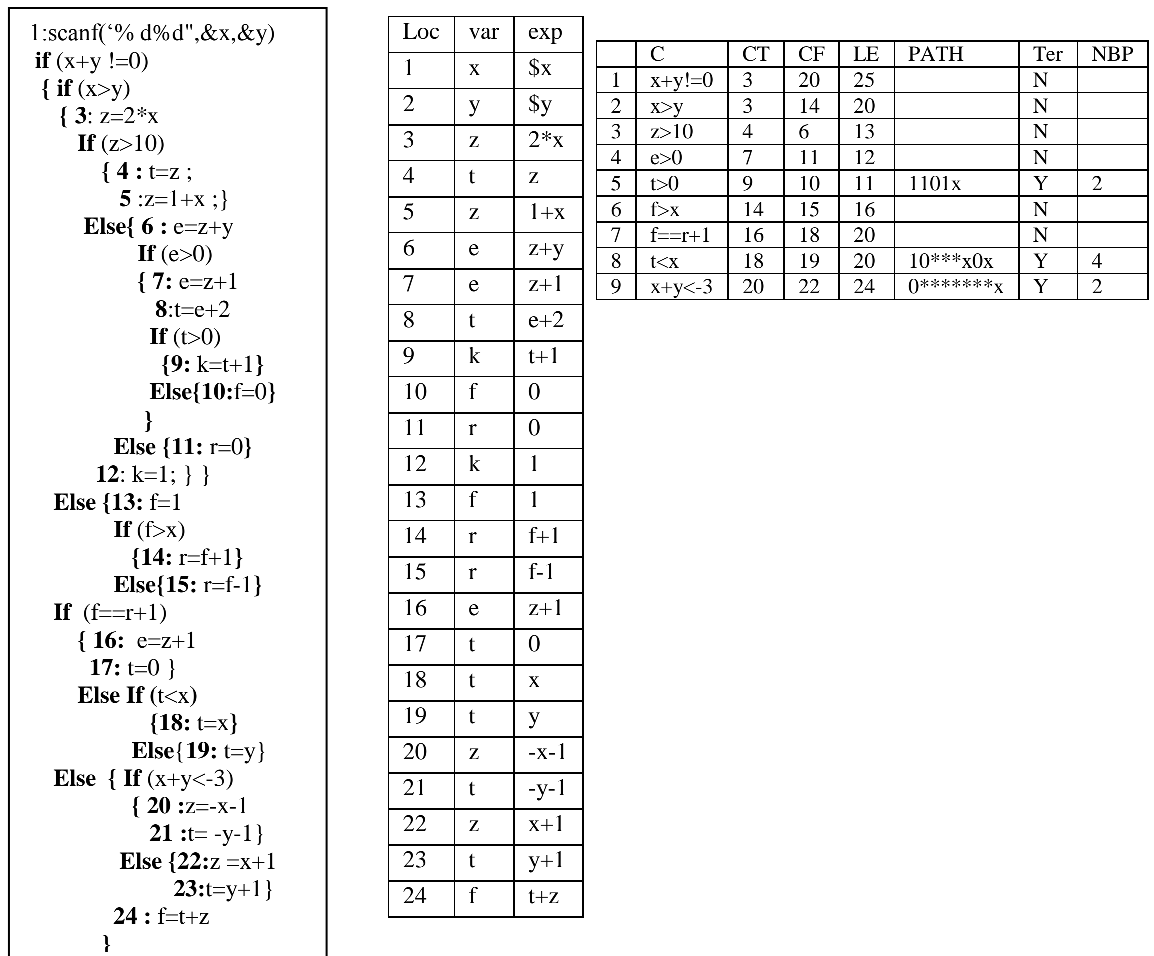
Input : Program : Prog, NbWorker;
Output: A set of test cases for Prog.
Initialization: NbPath=0;NBCond=0;TestCase=∅; (VT,CT)=Model(Prog)
For i=1 to CardCT
Do Begin **If** Terminal(CT[i])
  Then Path[i]=Compute_Path(i);
    NBP[i]=Compute_NBP(i);
    NBC[i]=Compute_NBC(i);
    NbPath=NbPath+NBP[i];
    NBCond=NBCond+NBC[i];
 End;
Divide (NbPath, NBcond, nbWorker);
For k=1 to NbWorker
Do Assign-test(k, Set);
IF (Receipt(k, Ck)) **Then** TestCase=TestCase∪Ck

Figure 4.4. Coordinateur

- **La fonction Compute_NBP ()** calcule, pour chaque élément de CT le nombre de chemins représentés par celui-ci.
- **La fonction Compute_NBC ()** calcule le nombre de conditions de chaque chemin.
- **La procédure Divise()** a pour rôle de diviser équitablement les chemins sur les nœuds selon le nombre total de chemins et de nœuds. Elle regroupe les chemins en ensembles, chacun sera attribué à un nœud.
- **La procédure Attribuer-test ()** assigne un ensemble de chemins à chaque nœud.
- **La fonction booléenne Réception (k, Ck)** est True si le nœud k a envoyé ses résultats Ck au coordinateur.

4.3.3. Etude de cas

Dans cette section, nous présentons un exemple illustratif comme étude de cas pour élucider notre approche et souligner sa pertinence. Nous supposons que nous disposons de deux nœuds :



```
1:scanf('% d%d",&x,&y)
if (x+y !=0)
 { if (x>y)
   { 3: z=2*x
     If (z>10)
        { 4 : t=z ;
           5 :z=1+x ;}
      Else{ 6 : e=z+y
            If (e>0)
            { 7: e=z+1
               8:t=e+2
              If (t>0)
                {9: k=t+1}
              Else{10:f=0}
              }
         Else {11: r=0}
      12: k=1; } }
  Else {13: f=1
         If (f>x)
            {14: r=f+1}
         Else{15: r=f-1}
  If  (f==r+1)
     { 16:  e=z+1
       17: t=0 }
     Else If (t<x)
              {18: t=x}
           Else{19: t=y}
  Else  { If (x+y<-3)
           { 20 :z=-x-1
              21 :t= -y-1}
          Else {22:z =x+1
                 23:t=y+1}
         24 : f=t+z
        }
```

Loc	var	exp
1	x	$x
2	y	$y
3	z	2*x
4	t	z
5	z	1+x
6	e	z+y
7	e	z+1
8	t	e+2
9	k	t+1
10	f	0
11	r	0
12	k	1
13	f	1
14	r	f+1
15	r	f-1
16	e	z+1
17	t	0
18	t	x
19	t	y
20	z	-x-1
21	t	-y-1
22	z	x+1
23	t	y+1
24	f	t+z

	C	CT	CF	LE	PATH	Ter	NBP
1	x+y!=0	3	20	25		N	
2	x>y	3	14	20		N	
3	z>10	4	6	13		N	
4	e>0	7	11	12		N	
5	t>0	9	10	11	1101x	Y	2
6	f>x	14	15	16		N	
7	f==r+1	16	18	20		N	
8	t<x	18	19	20	10***x0x	Y	4
9	x+y<-3	20	22	24	0*******x	Y	2

Figure 4.5. Etude de cas. Génération de cas de test dans le Cloud.

Dans le tableau, nous avons reporté uniquement les chemins terminaux. Pour obtenir une répartition bien équilibrée, une solution possible consiste à attribuer Path[5] et Path[9] à un même noeud (W1) et Path[8] à un autre noeud (W2).

Génération automatique des cas de test

Nœud W1

Path[5] = 1101x => Path1[5]=11010 et Path2[5] = 11011.

Path1[5]: C [1]≡T et C[2]≡T et C[3]≡F et C[4]≡T et C[5]≡F (I)

WP ([1,3[, x+y!=0)=($x+$y!=0)

WP([1,3[, x>y)=($x>$y)

WP([1,3[∪[3,3[, z<=10)=(2* $x<=10)

WP ([1,3[∪[3,3[∪[6,6[, e>0)=(2*$x+1>0)

WP ([1,3[∪[3,3[∪[6,6 [∪[7,9[, t<= 0)=(2 *$x +3<=0)

Ainsi, la formule (I) est la suivante :

($x+$y!=0)∧($x>$y)∧(2*$x<=10)∧(2*$x+1>0)∧(2*$x+3<=0)

Cette formule est insatisfiable donc le chemin chemin1[5] n'est pas faisable. En d'autres termes le bloc 5 n'est pas accessible.

Path2[5] : La formule est :

($x+$y!=0)∧($x>$y)∧(2*$x<=10)∧(2*$x+1>0)∧(2*$x+3>0)

Une solution possible est $x=4 et $y=2, qui constitue un test pour ce chemin. Un raisonnement similaire est effectué pour le chemin Path[9] (il ya 2 conditions à vérifier)

Nœud W2

Path[8]=10***x0x =>Path1[8]=10***000 ; Path2[8]=10***001;

Path3[8]=10***100 ; Path4[8]=10***101

Le même raisonnement est effectué pour calculer l'ensemble des cas de test couvrant les quatre chemins. Enfin, l'ensemble des cas de test de l'ensemble du programme est l'union de tous les cas de test calculés.

4.4. Discussion

Dans ce chapitre, nous avons présenté une nouvelle approche pour les tests symboliques distribués. Nous avons d'abord défini l'exécution symbolique arrière pour la méthode de génération de tests. Ensuite, nous avons proposé une version distribuée de notre solution permettant de tirer profit de la modélisation adoptée des programmes. Notre solution présente plusieurs avantages :

1. C'est une méthode arrière : Au lieu de l'exécution du programme dans son ensemble comme dans les autres méthodes, elle capture simplement l'impact de chaque instruction sur les prédicats considérés.
2. Notre solution distribuée est bien équilibrée sans nécessiter un partitionnement dynamique.
3. Il n’existe aucune dépendance entre les nœuds.
4. La communication entre les nœuds et le coordonnateur est minimale.

5. Chaque nœud est tenu d'avoir uniquement les informations concernant les chemins qu'il calcule.

Cependant, notre solution présente certains inconvénients, le plus important est la redondance. Ceci est dû au fait que nous avons choisi de maintenir la communication entre les nœuds minimale. Toutefois, la redondance pourrait être réduite si nous permettons aux nœuds de partager plus d'informations sur d'éventuelles parties communes des chemins à analyser.

Chapitre 5

Génération Automatique de Programmes par la Programmation Génétique

La programmation génétique est une méthode évolutionnaire, inspirée par l'évolution biologique permettant de produire des programmes qui effectuent des tâches prédéfinies par l'utilisateur. C'est un procédé de génération automatique de programmes pour exécuter des tâches spécifiques [BBC+06]. La programmation génétique utilise un algorithme génétique pour la recherche, dans un espace de programmes possibles, celui qui est « optimal » dans sa capacité à réaliser une tâche particulière.

5.1. Méthodologie de la programmation génétique

5.1.1. Principe

Habituellement, dans la programmation génétique, quatre étapes sont utilisées pour générer un programme résolvant de la meilleure façon qui soit un problème:

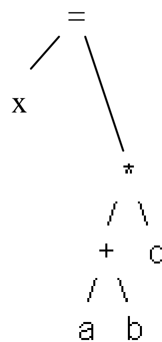
1. Générer une population initiale de compositions aléatoires de fonctions et de terminaux du problème.
2. Exécuter chaque programme de la population, avec tous les cas de Fitness et attribuer une valeur de fitness au programme en fonction de la façon dont il résout le problème. Les cas de fitness constituent un ensemble d'exemples de cas exhibant des entrées avec les sorties adéquates. Cet ensemble est utilisé pour évaluer les programmes générés. Il est à noter que cette phase est très coûteuse. En effet, si une population est constituée de 100 individus programmes, et que l'on dispose de 100 cas de Fitness, alors il est nécessaire d'effectuer 10000 exécutions pour évaluer tous les individus, puisque chaque individu doit être exécuté avec 100 cas de Fitness.
3. Selon leur valeur de Fitness, sélectionner des individus pour le croisement et la mutation.

4. Créer une nouvelle population de programmes.
   i) Reproduire les meilleurs programmes existants
   ii) Créer de nouveaux programmes par mutation.
   iii) Créer de nouveaux programmes par croisement.
5. Le meilleur programme créé dans toutes les générations est désigné comme le résultat de la programmation génétique [BB07].

La programmation génétique développe des programmes, généralement représentés dans la mémoire sous forme d'arbres. Les arbres sont évalués de façon récursive. Chaque nœud de l'arbre est un opérateur de fonction, et chaque nœud terminal représente un opérande.

5.1.2. Opérateurs Génétiques

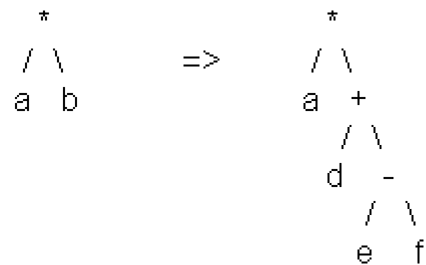
Les opérateurs génétiques utilisés pour les algorithmes génétiques sont aussi utilisés dans la programmation génétique. Cependant, leur mise en œuvre peut s'avérer plus difficile à cause des données qu'ils manipulent et de la représentation des individus par des arbres.
Exemple de représentation : L'affectation x= ((a+b)*c) est représentée par l'arbre:



- **Mutation**

Pour effectuer une mutation sur un individu, un nœud de l'arbre est choisi au hasard, le sous-arbre partant de ce nœud sera remplacé par un nouveau sous-arbre généré au hasard. On obtient ainsi un nouvel arbre représentant un nouveau programme.

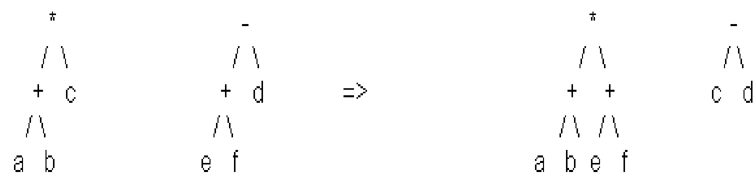
Exemple :



- ## Croisement

Le croisement entre deux individus, qui sont des programmes, est effectué en choisissant au hasard un nœud de l'arbre de chacun des programmes. Les sous arbres obtenus à partir de chacun de ces nœuds sont échangés d'un arbre à l'autre. On obtient ainsi deux nouveaux programmes.
Nous constatons que les opérations génétiques sont purement syntaxiques.

Exemple :



Une conséquence de la représentation des programmes par des arbres, est que traditionnellement la programmation génétique favorise l'utilisation des langages de programmation qui incarnent naturellement des structures arborescentes telles que les langages de programmation fonctionnels [BB07]. Les opérateurs génétiques sont conçus pour que la progéniture qui en résulte soit constituée d'individus syntaxiquement valides sans tenir compte des conséquences de l'application de cet opérateur sur les performances du nouvel individu. Par conséquent, quelques tentatives utilisant la sémantique ont récemment vu le jour, pour améliorer cet aspect. Parmi celles-ci l'utilisation des méthodes formelles.

5.2. Méthodes formelles pour la programmation génétique

Depuis ses débuts, plusieurs idées ont été proposées pour améliorer la programmation génétique. L'utilisation des méthodes formelles est une des tentatives qui ont été récemment mises en œuvre. Deux types de techniques formelles ont été utilisés en programmation génétique, l'interprétation abstraite [JN95] et la logique temporelle [GCP99]. L'interprétation abstraite effectue une analyse sur les domaines abstraits au lieu des domaines concrets. Par l'utilisation de l'interprétation abstraite, des informations intéressantes peuvent être déduites sur les propriétés de certains programmes. Dans [Joh02a, Joh0b], cette technique est utilisée comme une mesure de la fonction Fitness. Un problème de placement est étudié dans [Joh02a]. Avec ce genre de problème, il est très difficile d'utiliser des mesures de fitness traditionnelles qui sont basées sur l'ensemble des cas de fitness. En effet, tout d'abord, la génération d'un ensemble de cas de l'échantillon n'est pas facile dans cette situation. De plus, l'échantillon ne peut pas couvrir toutes les situations possibles. Dans [Kei03], l'interprétation abstraite est utilisée pour vérifier si un individu peut être défini dans l'ensemble des valeurs d'entrée. Par exemple, si un individu contient la fonction log (x) et que l'on peut déduire que la variable x peut prendre des valeurs négatives; cet individu sera considéré comme un individu indéfini, il doit être supprimé de la population. Dans [Joh07], un système est modélisé par un ensemble de formules de la logique temporelle. La fonction de Fitness est mesurée en comptant le nombre de formules satisfaites. L'individu qui satisfait le plus de propositions, aura une plus grande valeur de Fitness. L'inconvénient de cette approche est qu'une formule, qui est « presque » satisfaite, sera considérée comme une formule tout à fait insatisfaite. Cette faiblesse est considérée par certaines recherches plus tard dans [KP08a, KP08b], qui ont utilisé la programmation génétique avec la logique temporelle pour vérifier si un chemin dans le graphe qui représente le comportement du système, est satisfait par la formule. La fonction fitness est basée, dans ce cas, sur des scores en fonction du nombre de chemins satisfaits dans le graphe. L'avantage des méthodes formelles réside dans leurs fondements rigoureux, ce qui pourrait aider la programmation génétique à élaborer des programmes de plus en plus performants. Cependant, elles sont connues par leur complexité et sont difficiles à mettre en œuvre, ce qui explique qu'elles ont été utilisées principalement pour des mesures de Fitness.

5.3. Un nouveau cadre pour la programmation génétique multi-objectif [AK12a]

5.3.1. Principe et motivations

Le but de la programmation génétique multi-objectif, est de générer automatiquement un programme calculant un ensemble de résultats. Dans ce qui suit, nous présentons un nouveau cadre pour la programmation génétique multi-objectif. Tout d'abord, notre approche intervient au niveau de la représentation des programmes. En effet, puisque les individus doivent être intensivement manipulés par : les évaluations, les croisements et les mutations, il est essentiel d'adopter une modélisation des programmes qui soit aussi simple que performante. Nous adoptons la même représentation des individus programmes que dans les chapitres précédents. Chaque individu de la population est représenté par deux tables. La table des variables, VT, enregistre les différentes expressions permettant le calcul des valeurs des variables et la table CT décrivant la structure de celui-ci. Notre objectif est d'une part de faciliter la création, l'exécution et l'amélioration des individus, et d'autre part d'effectuer des croisements sémantiques : i.e. des croisements basés sur la sémantique et non uniquement sur la syntaxe. La modélisation adoptée permet de faire des exécutions symboliques de manière à réduire considérablement le temps d'exécution des programmes et de permettre le bien-fondé des croisements de programmes. Elle permet entre-autres :

1. Une manipulation aisée des programmes.
2. Une évaluation rapide de la population.
3. Des croisements de programmes "Sémantiquement justifiés".
4. Une amélioration effective de la population à chaque nouvelle itération.

Dans notre approche, nous utilisons les étapes suivantes : Génération d'une population initiale, évaluation des individus, et amélioration des individus. Ces étapes sont présentées dans les sections suivantes.

5.3.2. Population initiale

La population initiale est constituée d'un ensemble de doublets (VTi, CTi), une paire pour chaque individu. Les tables sont remplies de manière aléatoire. Les fonctions du problème et les terminaux sont utilisés à la fois

dans la table des variables et de contrôle. Lors du remplissage des tables (VTi, CTi), les seules contraintes à vérifier sont des contraintes syntaxiques : Exemple : LE>LT...

5.3.3. Evaluation des individus

Pour évaluer un individu, il est nécessaire de l'exécuter avec tous les cas de Fitness. Afin de réduire le temps d'exécution des individus, et de tirer profit de quelques propriétés des exécutions, nous procédons à des exécutions symboliques. L'idée est de calculer une formule pour chaque sortie du programme génétique : nous l'appelons *expression du résultat*. Ainsi, pour chaque variable de sortie, l'expression du résultat correspondante résume toutes les expressions des variables de sortie en fonction des variables d'entrée. Par conséquent, pour un individu, la même formule est utilisée pour évaluer tous les cas de Fitness en remplaçant les variables par leurs valeurs d'entrée, et en vérifiant si la valeur de sortie correspondante est correcte. Pour effectuer des exécutions symboliques, nous utilisons la plus faible précondition de la même manière que précédemment.

5.3.3.1 Exécution symbolique des individus

Soit un individu Progi de la population, Progi = (VTi, CTi). L'évaluation de Progi est réalisée comme suit :

1. Pour chaque variable en sortie o_k, introduire une variable de résultat R_i, ajouter dans VTi et CTi les lignes correspondant à l'instruction suivante si ($R_{ik}=o_k$) alors $o_k = R_{ik}$ (qui n'a aucun effet sur l'exécution de Progi).
2. Calculer les préconditions successives du prédicat ($R_{ik}=o_k$) relativement à toutes les lignes de CTi à partir de la dernière instruction jusqu'au début.
3. L'expression $ExpR_{ik}$ qui en résulte représente, toutes les expressions possibles de la sortie o_k avec leurs conditions correspondantes.
4. Pour chaque cas de Fitness, remplacer les variables d'entrée dans $ExpR_{ik}$. par leur valeur et en déduire la valeur de R_{ik} rendant $ExpR_{ik}$ vraie. Ainsi, la même expression ExpRik est utilisée pour calculer tous les cas de Fitness pour l'individu Progi. Ceci constitue le premier avantage des exécutions symboliques.

5.3.3.2 Exemples

Pour illustrer l'idée précédemment exposée, nous introduisons deux exemples d'individus : Le premier est un programme simple sans répétition, le second contient une boucle.

Exemple 1 : Evaluation d'un individu ne contenant pas de répétition

Soit Progi le programme ayant deux variables en entrées : x et y, et une variable de sortie b. Appelons r l'expression du résultat, correspondant à l'unique variable de sortie b :

1. Ajout de la ligne 5 dans VT et L3 dans CT.

Loc	Var	Exp
1	A	X
2	A	Y
3	B	a+1
4	B	a-1
5	**R**	**B**

	C	LT	LF	LE
L1	x>y	1	2	3
L2	a>0	3	4	5
L3	**r=b**	**5**	**-1**	**6**

2. Calcul des Préconditions

$WP(L2,r=b)$

$$=(a>0)\wedge WP([3,4[,r=b)\vee(a<=0)\wedge WP([4,5[,r=b)$$

$$=(a>0)\wedge WP(3,r=b)\vee(a<=0)\wedge WP(4,r=b)$$

$$=(a>0)\wedge(r=a+1)\vee(a<=0)\wedge(r=a-1)$$

$$=D$$

$$WP(L1,D)=(x>y)\wedge WP([1,2[,D)\vee(x<=y)\wedge WP([2,3[,D)$$

$$=(x>y)\wedge WP(1,D)\vee(x<=y)\wedge WP(2,D)$$

$$=(x>y)\wedge[(x>0)\wedge(r=x+1)\vee(x<=0)\wedge(r=x1)]\vee$$

$$(x<=y)\wedge[(y>0)\wedge(r=y+1)\vee(y<=0)\wedge(r=y-1)] = \mathbf{(I)}$$

L'expression (I) est l'expression du résultat. Elle représente exactement le résultat r exprimé en fonction de toutes les valeurs possibles des variables d'entrée x et y. L'évaluation de Progi sur tous les cas de Fitness consiste à remplacer x, y et r par leurs valeurs et de vérifier si (I) est vrai. Soient C1 et C2 deux cas de Fitness C1= (x=10, y=1, b=5) et C2 = (x=-5, y=6, b=7). Exécutons Progi sur ces deux cas :

Pour C1: nous remplaçons dans (I) : x par 10 et y par 1.

$$(I) = (10>1)\wedge[(10>0)\wedge(r=10+1)\vee(10<=0)\wedge(r=10-1)]\vee (10<= 1)\wedge[(1> 0)\wedge(r=1+1)\vee(1<=0)\wedge(r=1-1].$$
$$= (T)\wedge[(T)\wedge(r=11)\vee(F)\wedge(r=9)]\vee(F)\wedge[(T)\wedge(r=2)\vee(F)\wedge(r=0)].$$
$$= (r = 11).$$

Cette expression est vraie si et seulement si r = 11. Ainsi, le résultat trouvé par l'exécution de l'individu Progi est 11 tandis que le résultat attendu est b = 5.

Pour C2 :

$$(I) = (-5>6)\wedge[(-5>0)\wedge(r=1-5)\vee(-5<=0)\wedge(r=-5-1)]\vee (-5<= 6)\wedge[(6>0)\wedge(r=6+1)\vee(6<=0)\wedge(r=6-1)].$$
$$= (F)\wedge[(F)\wedge(r=-4)\vee(T)\wedge(r=-6)]\vee(T)\wedge[(T)\wedge(r=7)\vee(F)\wedge(r=5].$$

= (r = 7) qui représente le résultat escompté (b=7).

Donc, pour chaque individu, la même formule est utilisée pour calculer les résultats correspondant à tous les cas de Fitness.

Exemple 2 : Evaluation d’un individu programme contenant une boucle

Soit un individu ayant une entrée: n ; et une sortie: s, nous disposons des cas de Fitness: (n = 3, s = 5) :

1. Ajout de la ligne 5 dans VT et L2 dans CT

```
1:  i=1;
2: s=0;
   while(i<n)
3: { s=s+i;
4:  i=i+1}
   if( r==s)
5: r=s
```

Loc	Var	Exp
1	I	1
2	S	0
3	S	s+i
4	I	i+1
5	**R**	**S**

	Cd	LT	LF	LE
L1	i<n	3	-2	5
L2	**r=s**	**5**	**-1**	**6**

2. Calcul des préconditions

WP (L1, r = s) =?

C0 = (i <3); P0 = (r = s), k = 1;

P1 = WP ([3,5 [, r = s) = (s + r = i) ; C1 = WP([3,5 [, i<n)=(i +1<n)

Ainsi, C1 '= (1 +1 <3) = (2 <3) = True. D’où:
P2=WP([3,5[,r=s+i)=(r=s+i+i+1); C2=WP([3,5[,i+1<n)=(i+1+1<n)
C2'=(3<3)=False.

Donc, WP (L1, r = s) = P2 = (r = 3). Ainsi, le résultat calculé par l'individu est 3, alors que le résultat attendu est 5.

5.3.4. Amélioration des individus

5.3.4.1 Motivations

Habituellement, dans la programmation génétique, certains programmes de la population sont sélectionnés pour être améliorés. Les programmes sont élus en fonction de leur valeur de Fitness. Cependant, les mutations et les combinaisons des programmes sont généralement effectuées de manière arbitraire. Par conséquent, il n'y a pas d’explication "sémantique" justifiant les questions suivantes:

• Pourquoi est-il approprié d’effectuer ce type de mutation sur cet individu?
• Pourquoi effectuer un croisement sur ces deux individus ?
• Pourquoi le croisement sur les deux individus sélectionnés devrait-il être effectué de cette façon?

Nous ne nous intéresserons pas au premier point, mais au croisement des programmes. Notre but est de tenter de réaliser des croisements "sémantiquement justifiés". Pour atteindre cet objectif, nous exploitons les informations déduites des exécutions symboliques. Tout d'abord, soulignons les points suivants :

1. Nous nous situons dans le contexte multi-objectif : Le problème pour lequel nous souhaitons générer un programme a plusieurs variables de sortie : o_1... o_m nous appelons notre opérateur de croisement : Multi-crossover.

2. Nous notons Fit_{ik} la valeur de Fitness du programme $Prog_i$ pour le calcul de La sortie o_k. Donc, dans notre approche, la fonction fitness n'est pas quantifiée par une valeur unique. Cela se justifie par le fait qu'un programme peut calculer une variable de sortie o_t de façon très performante mais ne parvient pas à calculer de façon

satisfaisante une autre sortie o_s. Ainsi, les programmes sont jugés relativement à chaque variable de sortie indépendamment des autres, ce qui n'est pas habituel dans la programmation génétique.

3. Pour un individu $Prog_i$, nous appelons S_i l'ensemble de toutes ses expressions de résultat : une expression de résultat par sortie : $S_i = \{ExpRes_{ir},, k = 1 .. m\}$.

4. Soit F une formule logique du premier ordre. F est dite en forme exclusive si l'une des deux conditions est vérifiée :

   - F est une formule atomique (ne contient pas $\wedge$ ni $\vee$)
   - F est de la forme $C \wedge P \vee \neg C \wedge Q$, où P et Q sont deux formules sous forme exclusive.

Remarquons que les calculs des plus faibles préconditions fournissent toujours comme résultat une formule sous forme exclusive. Ce qui implique que les expressions $ExpRes_{ik}$ sont toutes en forme exclusive.

5.3.4.2 Multi-Crossover

Notre objectif est de réaliser un croisement judicieux des programmes. Nous évaluons les programmes par rapport aux variables de sortie. Par conséquent, le croisement des programmes est effectué relativement à une variable précise de sortie et non à toutes les variables comme il est d'usage dans la programmation génétique. Nous devons tirer profit de chaque Progi ayant une bonne valeur de Fitness. C'est pourquoi notre opérateur de croisement ne sert pas à créer deux nouveaux programmes, mais conserve le premier programme (le meilleur) et ne modifie que le second. Le premier programme pourrait à son tour tirer profit d'un autre programme qui est meilleur pour le calcul d'une autre sortie. Les exécutions symboliques rendent possible l'isolement des instructions relatives aux calculs d'une sortie considérée. Donc, l'idée de notre opérateur de croisement consiste à remplacer, pour deux programmes $Prog_i$ et $Prog_j$ tels que $Prog_i$ est meilleur que $Prog_{j,}$ dans le calcul de la sortie o_k, c'est à dire $Fit_{ik} > Fit_{jk}$, l'expression du résultat $ExpRes_j$ par $ExprRes_i$ dans S_j,où dans le calcul de la sortie o_k. L'ensemble obtenu S_j' est ensuite traduit en un nouveau programme $Prog_{j'}$ où toutes les sorties o_r telles que $r \neq k$ sont calculées comme dans $Prog_j$, et

o_k est calculé de la même manière que réalisé par $Prog_i$. Ceci constitue le deuxième avantage de l'utilisation des exécutions symboliques plutôt que des exécutions réelles. L'algorithme de la figure 5.1. décrit le fonctionnement de l'opérateur de croisement, où $Prog_i$ et $Prog_j$ sont deux programmes. Nous notons $Prog_i \bowtie_r Prog_j$ le croisement de $Prog_i$ et $Prog_j$ relativement à la variable de sortie o_r.

Algorithme: Crossover($Prog_1$,$Prog_2$,o_r)

1. **Input**: S_1, S_2, o_r
2. **Output**: $Prog_2'=(VT_2',CT_2')$
3. $S_2=S_2-\{ExpRes_{2r}\}\cup\{ExpRes_{1r}\}$
4. **For** each formula $ExpRes_{2k}$ in S_2
5. **Do** Translate ($ExpRes_{2k}$);
6. **End.**

Figure 5.1. Algorithme calculant le croisement de deux programmes

Ayant les expressions des résultats de toutes les variables de sortie, la prochaine étape consiste à déduire un programme qui les réalise. La traduction d'une formule en forme exclusive F en un individu = Progi (VTi, CTi) est réalisée par l'algorithme décrit sur la figure 5.2.

Algorithme: Translate(F)

1. **Input**: F: An EF Formula.
2. **Output**: VT,CT

3: **If** F is an atomic formula of the form v=exp
4: Then Let o be the output variable corresponding to v
5: Insert(o,exp) in the current line of VT
6: Else F of the form $C\wedge P\vee\neg C\wedge Q$
7: Let LT be the current line in VT.
8: insert(C,LT,LF,LE) in CT
9: Translate (P);
10: Translate(Q)
End.

Figure 5.2. Algorithme construisant VT et CT à partir d'une formule exclusive.

Dans la ligne (8), les valeurs de LF et de LE ne sont pas encore connues. Elles seront mises à jour respectivement dans les lignes (9) et (10). La traduction d'un ensemble de formules est effectuée en traduisant chaque formule, indépendamment des autres. L'ordre dans lequel les traductions sont effectuées n'est pas important, car dans les expressions des résultats, chaque sortie est exprimée uniquement à l'aide des variables d'entrée.

Exemple: Traduisons l'expression (I) de l'exemple 1:

$$(I) = (x>y)\wedge[(x>0)\wedge(r=x+1)\vee(x<=0)\wedge(r=x-1)]\vee$$
$$(x<=y)\wedge[(y>0)\wedge(r=y+1)\vee(y<=0)\wedge(r=y-1)]$$

l'individu obtenu est :

Loc	Var	Exp
1	B	x+1
2	B	x-1
3	B	y+1
4	B	y-1

C	LT	LF	LE
x>y	1	3	5
x>0	1	2	3
y>0	3	4	5

La variable r correspond à la variable de sortie b. On remarque que les tables VT et CT sont différentes des tables initiales. En effet, dans les tables initiales nous avons utilisé une variable intermédiaire *a* qui a disparu dans (I), puisque dans (I), nous avons juste les sorties en fonction des variables d'entrées.

5.3.4.3 Exemple d'application de l'opérateur de croisement

Considérons une population constituée de trois programmes : PROG1, PROG2 et PROG3. Nous supposons que notre problème a quatre variables d'entrée a, b, c, et d, et trois variables de sortie o1, o2, et o3. o1=max (a,b)*c*d ; o2=|a*c|-bd ; o3=min (a, c) + max (b, d). Il est clair que ces fonctions ne sont pas connues dans le problème. Les seules informations dont on dispose sont les cas de Fitness pour évaluer les individus. Les trois individus sont les suivants :

Prog1

Loc	Var	Exp
1	mx	A
2	Mx	B
3	o1	mx*c*d
4	o2	a*c-b-d
5	Mn	C
6	Mn	A
7	o3	mn+mx
8	res1	o1
9	res2	o2
10	res3	o3

Cd	LT	LF	LE
a>b	1	2	3
a>c	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

$ExpRes_{11}$ = (a>b)∧(res1=a*c*d)∨(a<= b)) ∧ (res1 = b * c * d)

$ExpRes_{12}$ = (res2=a*c-b-d)

$ExpRes_{13}$ = (a>c)∧(a>b) ∧ (res3=c+ a) ∨ (a <= b) ∧ (res3 = c + b)) ∨
(a<= c) ∧ (a> b) ∧ (res3 = 2 * a) ∨ (a <= b) ∧ (res3 = a + b))

Prog2

Loc	Var	Exp
1	o1	a*c*d
2	Abs	a*c
3	Abs	-a*c
4	o2	abs-b-d
5	M	B
6	M	D
7	o3	m+a
8	res1	o1
9	res2	o2
10	res3	o3

CD	LT	LF	LE
a*c>0	2	3	4
b>d	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

$ExpRes_{21}$ = (res1=a*c*d)

$ExpRes_{22}$ = (a*c> 0) ∧ (res2 = a * c-d-b) ∨ (a*c <= 0) ∧ (Res2 =-a * c-d-b)

$ExpRes_{23}$ = (b> d) ∧ (res3 = b + a) ∨ (b <= d) ∧ (res3 = d + a)

Prog 3

Loc	Var	Exp
1	o1	b*c*d
2	o2	c-b-d
3	Mi	A
4	Mi	C
5	Ma	B
6	Ma	D
7	o3	mi+ma
8	res1	o1
9	res2	o2
10	res3	o3

CD	LT	LF	LE
a<c	3	4	5
b>d	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

$ExpRes_{31}$ = (res31 = b * c * d);
$ExpRes_{32}$ = (res2 = c-b-d)
$ExpRes_{33}$ = (b> d) ∧ ((a = c <c)∧ (res3=a+b)∨(a>) ∧ (res3 = c + b)) ∨
(b <= d) ∧ ((a = c <c)∧ (res3=a+d)∨(a>) ∧ (res3 = c + d))

Dans un problème réel, nous n'avons pas l'expression des fonctions attendues, mais nous avons des cas de Fitness. Ainsi, supposons qu'après l'exécution de tous les cas de Fitness, nous constatons que chaque individu $Prog_i$ est le meilleur dans le calcul de la sortie o_i. Par conséquent, calculons $Prog_1 \bowtie_1 Prog_2 = (Prog_1, Prog_2')$
Nous aurons alors les expressions des résultats suivantes pour $Prog_2$:
$ExpRes_{21}' = ExpRes_{11}$;
$ExpRes_{22}' = ExpRes_{22}$;
$ExpRes_{23}' = ExpRes_{23}$

Maintenant, calculons $PROG_3 \bowtie_3 Prog_2' = (Prog_3, Prog_2'')$. Nous aurons alors les expressions des résultats suivantes ExpRes pour $Prog_2''$:

$ExpRes_{21}$"= $ExpRes_{11}$; $ExpRes_{22}$" = $ExpRes_{22}$; $ExpRes_{23}$"= $ExpRes_{33}$. Finalement, La traduction de cet ensemble d'expressions de résultat donne :

Loc	Var	Exp
1	o1	a*c*d
2	o1	b*c*d
3	o2	a*c-b-d
4	o2	-a*c-b-d
5	o3	a+b
6	o3	c+b
7	o3	a+d
8	o3	c+d

CD	LT	LF	LE
a>b	1	2	3
a*c>0	3	4	5
b>d	5	7	9
a<c	5	6	7
a<c	7	8	9

qui correspond au programme C suivant :

```
If  (a>b)
      o1=a*c*d
else o1= b*c*d
if (a*c>0)
       o2=a*c-b-d
else  o2= -a*c-b-d
if (b>d)
    if (a<c)
          o3=a+b
    else o3=c+b
else if (a<c)
            o3=a+d
      else o3=c+d
```

Nous remarquons que ce programme, généré automatiquement à partir $PROG_1$, $PROG_2$ et $PROG_3$, calcule correctement les trois sorties o_1, o_2, et o_3. Nous constatons également que le programme résultant ne se résume pas en une transformation syntaxique d’aucun des programmes initiaux.

5.4. Discussion

Nous avons présenté une approche originale et efficace permettant d'améliorer la programmation génétique multi-objectif. La modélisation que

nous avons utilisée est aussi simple qu'efficace. Notre méthode présente plusieurs contributions:

1. Le calcul de la valeur de Fitness relativement à chaque sortie est aussi réaliste qu'avantageux. En effet, un même programme peut être performant dans le calcul de certains résultats, mais inadéquat pour d'autres.
2. La traduction de l'expression du résultat d'un programme est la façon la plus simple de garantir que le programme construit calcule, comme nous le souhaitons, la sortie correspondante. En outre, la traduction se fait sans difficultés considérables.
3. Notre opérateur de croisement garantit que le programme obtenu est effectivement meilleur que son prédécesseur.
4. Toutes les expressions du résultat doivent être calculées pour évaluer les cas de Fitness. Par conséquent, l'opérateur de croisement ne nécessite pas de calculs supplémentaires.
5. Les calculs en arrière pourraient être considérés comme une technique d'abstraction puisque dans une investigation en arrière nous suivons juste les variables souhaitées.
6. Tous les extra-traitements, i.e. les traitements qui ne contribuent dans le calcul d'aucune sortie, disparaîtront dans la solution finale, car ils n'apparaissent pas dans les expressions des résultats.

Conclusion Générale

L'exécution symbolique des programmes est une technique très puissante dans le domaine de l'analyse des programmes. Toutefois, en dépit des résultats très satisfaisants obtenus par l'application de cette technique notamment dans le domaine du test de programme, celle-ci souffre d'un obstacle très contraignant : l'explosion du nombre de chemins d'exécution. Cet obstacle a engendré une limitation quant à l'utilisation de cette technique dans d'autres domaines. Ce travail est une tentative pour remédier à cet obstacle. Notre contribution démarre de la constatation suivante : La totalité des travaux réalisés dans le domaine de l'application de l'exécution symbolique au test et à la vérification des programmes, procèdent de façon descendante : le programme est représenté par le graphe de flot de contrôle (CFG), et l'analyse démarre à partir de la première instruction du programme et procède par calculs successifs des plus fortes post-conditions pour avancer dans le programme. A notre sens, c'est ce mode de parcours qui accentue le problème d'explosion de chemins. En effet, par exemple pour la vérification de propriété de sûreté, au lieu d'exécuter tous les chemins possibles sans être sûr qu'ils aboutissent à l'erreur, il serait plus judicieux de démarrer de la localisation d'erreur, et d'essayer de retrouver les conditions sous lesquelles cette localisation est accessible, en calculant des préconditions au lieu des post-conditions. Nous définissons ainsi une analyse orientée but. Il est clair que la représentation des programmes sous forme d'arbres n'est pas adéquate à ce genre de parcours. Pour cela, nous avons défini une autre représentation qui est efficace tout en étant très simple. De plus, cette représentation permet une analyse compositionnelle. Nous avons appliqué notre approche à la vérification des propriétés de sûreté des programmes, au test de logiciel, et à la génération automatique des programmes. Vu que le test des programmes nécessite une exploration plus vaste des chemins d'exécution (selon la couverture souhaitée), nous avons défini une version parallèle et distribuée de notre analyse.

Pareillement au domaine de l'analyse des programmes, en programmation génétique aussi les programmes sont représentés par des arbres (syntaxiques abstraits). Par conséquent, l'amélioration des individus manipule des arbres de façon purement syntaxique. Ainsi, un croisement consiste à considérer un nœud et à permuter les branches des deux individus : aucune justification sémantique n'est avancée. De plus, l'évaluation de la population nécessite l'exécution de tous les individus avec tous les cas de

test. Ceci entrave le passage à l'échelle de la programmation génétique. Pour remédier principalement à ces deux inconvénients, nous avons développé un nouveau cadre pour la programmation génétique multi-objectif. Celui-ci permet une exécution rapide des individus, une évaluation de chaque programme relativement à chaque résultat, et un croisement sémantiquement justifié.

Ceci étant, beaucoup de perspectives s'ouvrent à ce travail. Nous pouvons les résumer en deux grandes classes :

1. Amélioration de la méthode :

a. Génération automatique des invariants.
b. Adaptation de l'approche de vérification à la programmation orientée objet : code source et bytecode java.
c. Passage des tests aux preuves et inversement.
d. Hybridation de l'exécution symbolique avec d'autres méta-heuristiques.
e. Utilisation de Datalog dans la méthode, pour la génération automatique des cas de test.
f. Application du modèle MapReduce pour la génération automatique des cas de test.

2. Application de la méthode à divers domaines :

a. Evaluation de la nouvelle méthode de programmation génétique dans un domaine approprié.
b. Modélisation , vérification, et génération automatique de cas de test pour les systèmes multi-agent.
c. Génération automatique des cas de test pour les applications web.
d. Génération automatique des cas de test pour les applications base de données.
e. Synthèse de programmes.
f. Utilisation de l'exécution symbolique pour la sécurité du logiciel : transformation pertinente de programmes, obfuscation,…

Références Bibliographiques

[AL95] Abadi M. and Lamport L., Conjoining specifications. ACM Transactions on Programming Languages and Systems 17, 3, 507–534. (1995)

[ACF08] Alba E., Chicano F., Ferreira M., Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms. ACM GECCO '08 Atlanta, Georgia, USA. (2008)

[ASR10] Aditya N., Sriram V., Rajamani S., An Empirical Study of Optimizations in Yogi . In ICSE '10: International Conference on Software Engineering, May (2010)

[AK12a] Aleb N. and Kechid S., A New Framework for a scalable genetic programming. ACM GECCO'12 Philadelphia (2012)

[AK12b] Aleb N. and Kechid S., Path coverage testing in the Cloud IEEE ICCIT Tunisia. (2012)

[ATK11] Aleb N., Tamen,Z., Kamel N., An evolutionary approach for program verification Medi11. Lecture notes in computer science Springer. (2011)

[ATK13] Aleb N., Tamen Z., Kamel N., Toward a backward model-checking. IJCAET, Volume 5, Number 1, inderscience. (2013)

[AH99] Alur R. and Henzinger T., Reactive modules. Formal Methods in System Design 15, 1, 7–48. (1999)

[AHM+98] Alur R., Henzinger T., Mang F., Qadeer S., Rajamani S., and Tasiran S., Mocha: Modularity in model checking. In CAV 98: Computer-Aided Verification. Lecture Notes in Computer Science 1427. Springer-Verlag, 521–525. (1998)

[AGT08] Anand S., Godefroid P. and Tillmann N., Demand-Driven Compositional Symbolic Execution. In TACAS, vol. 4963 of LNCS, pp. 367-381, Springer, (2008)

[BBK+10] Ball T., Bounimova E., Kumar R., Levin V., Slam2: Static Driver Verification with Under 4% False Alarms. FMCAD. (2010)

[BMR01] Ball T., Majumdar R., Rajamani S., Automatic predicate abstraction of C programs. In SIGPLAN Conference on Programming Language Design and Implementation, pp.203-213. (2001)

[BBC+06] Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S. and Ustuner A. Thorough static analysis of device drivers. In EuroSys. 73–85. (2006)

[BMM+01] Ball T., Majumdar R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. In PLDI 01: Programming Languages Design and Implementation. ACM, 203–213. (2001)

[BMR05] Ball T., Millstein T. and Rajamani S., Polymorphic predicate abstraction. ACM Transactions on Programming Languages and Systems 27, 2, 314–343. (2005)

[BPR01] Ball T., Podelski A. and Rajamani S. Boolean and Cartesian abstractions for model checking C programs. In TACAS 01: Tools and Algorithms for Construction and Analysis of Systems. Lecture Notes in Computer Science 2031. Springer-Verlag, 268–283. (2001)

[BPR02] Ball T., Podelski A. and Rajamani S.Relative completeness of abstraction refinement for software model checking. In TACAS 02: Lecture Notes in Computer Science . Springer-Verlag, 158–172. (2002)

[BR02a] Ball T. and Rajamani S.Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research. (2002)

[BR02b] Ball T. and Rajamani S. The SLAM project: debugging system software via static analysis. In POPL 02: Principles of Programming Languages. ACM (2002)

[BR00a] Ball T. and Rajamani S., Bebop: A symbolic model checker for Boolean programs. In SPIN 00: SPIN Workshop. Lecture Notes in Computer Science 1885. Springer-Verlag, 113–130. (2000)

[BR00b] Ball T. and Rajamani S., Boolean programs: a model and process for software analysis. Tech. Rep. MSR Technical Report 2000-14, Microsoft Research. (2000)

[BBC+06] Banzhaf W., Beslon G.,Christensen S.,Foster J.,Kepe F., Lefort F.,Miller J.,Radman M., and Ramsden J., From artificial evolution to computational evolution. A research agenda. Nat. Rev. Genet.7(9), 729–735. (2006)

[BH09] Bardin S.and Herrmann P., Pruning the Search Space in Path-Based Test Generation. In ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, (Washington, DC, USA), pp. 240-249, IEEE Computer Society, (2009)

[BH08] Bardin S. and Herrmann P., Structural testing of executables. In ICST, pages 22–31. IEEE Computer Society, (2008)

[BJ08] Beadle L. and Johnson C.G., Semantically driven crossover in genetic programming. In Proceedings of the IEEE World Congress on Computational Intelligence, pages 111–116. IEEE Press, (2008)

[BNR+08] Beckman N., Nori A.,Rajamani S. and Simmons R., Proofs from tests. In ISSTA 08: International Symposium on Software Testing and Analysis. ACM, 3–14. (2008)

[BCH+04] Beyer D., Chlipala A., Henzinger T., Jhala R.,and Majumdar R., Generating tests from counterexamples. In ICSE 04: International Conference on Software Engineering. ACM, 326–335. (2004)

[BHJ+07] Beyer D., Henzinger T., Jhala R., and Majumdar R.,The software model checker Blast. Software Tools for Technology Transfer 9, 5-6, 505–525. (2007)

[BHT07] Beyer D., Henzinger T. and The´oduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis. In CAV 07: Computer- Aided Verification. Lecture Notes in Computer Science 4590. Springer-Verlag, 504–518. (2007)

[BCH+04] Beyer D., Chlipala A., Henzinger T., Jhala R., Majumdar R., The Blast query language for software verification. In: Proc.SAS, LNCS, vol. 3148, pp. 2–18. Springer, Berlin. (2004)

[BHJ+05] Beyer D., Henzinger T., Jhala R., Majumdar, R., Checking memory safety with Blast. In: Proc. FASE, LNCS, vol. 3442, pp. 2–18. Springer, Berlin (2005)

[BCG+09] Beyer D., Cimatti A., Griggio A., Keremoglu M., Sebastiani R., Software Model Checking via Large-Block Encoding. (2009)

[BCC+99] Biere A., Cimatti A., Clarke E., Fujita M. and Zhu Y., Symbolic model checking using SAT procedures instead of BDDs. In DAC 99: Design Automation Conference. ACM, 317–320. (1999)

[BCE08] Boonstoppel P., Cadar C., and Engler D., RWset: Attacking Path Explosion in Constraint-Based Test Generation. in TACAS 08 , pp. 351-366. (2008)

[BFH90] Bouajjani A., Fernandez J. and Halbwachs N., Minimal model generation. In CAV 90: Computer-aided Verification. Lecture Notes in Computer Science 531. Springer-Verlag, 197–203. (1990)

[BBC+05] Bozzano M., Bruttomesso A. , Cimatti T., Junttila S., Ranise P., Rossum V. and Sebastiani R., Efficient satisfiability modulo theories via delayed theory combination. In Computer Aided Veri cation, pp. 335-349, Springer. (2005)

[BB07] Brameier M. and Banzhaf W. Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation. Springer, Berlin. (2007)

[BDG+04] Brat G., Drusinsky D., Giannakopolou D., Goldberg A., Havelund K., Lowry M., Pasareanu C., Venet A., Washington R., and Visser W. Experimental evaluation of verification and validation tools on Martian Rover software. Formal Methods in Systems Design 25. (2004)

[BCF+08] Bruttomesso R., Cimatti, A. Franz E., Griggio A. and Sebastiani R., The Mathsat 4 SMT solver. In Computer Aided Verification, pp. 299-303, Springer. (2008)

[BGM06] Botella B., Gotlieb A., and Michel C., Symbolic execution of floating-point computations. Software Testing, Verification and Reliability, 16(2):97–121. (2006)

[BUZ+11] Bucur S., Ureche C., Zamfir, and Candea G., Parallel symbolic execution for automated real-world software testing," in 6th ACM SIGOPS/EuroSys, (2011)

[Bry86] Bryant R., Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35, 8, 677–691. (1986)

[BLS02] Bryant R., Lahiri, Seshia., Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions". Computer Aided Verification 106–122. (2002)

[BS08] Burnim, J. and Sen K.., Heuristics for Scalable Dynamic Test Generation,. In Automated Software Engineering,. ASE08. 23rd IEEE/ACM International Conference on, pp. 443-446. (2008)

[CBG09] Charreteur F.,Botella B., and Gotlieb A., Modelling dynamic memory management in constraint-based testing. Journal of Systems and Software (JSS), 82(11) :1755–1766. (2009)

[CGP+08] Cadar C., Ganesh V., Pawlowski P., Dill D., and Engler D., EXE: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC), vol. 12, no. 2, p. 10, (2008)

[CE05] Cadar C. and Engler D., Execution generated test cases: How to make systems code crash itself. Lecture notes in computer science, vol. 3639, p. 2, (2005)

[CFS09] Chandra S, Fink S.J, Sridharan M., Snugglebug: A Powerful Approach To Weakest Preconditions. PLDI'09, June 15–20,Dublin, Ireland. ACM (2009)

[CKC11] Chipounov V., Kuznetsov V. and Candea, G., S2E: A platform for in-vivo multi-path analysis of software systems. In Intl. Conference on Architectural Support for Programming Languages and Operating Systems . (2011)

[CCG+03] Chaki S., Clarke E. M., Groce A., and Strichman O. Predicate abstraction with minimum predicates. In CHARME. 19–34. (2003)

[CCG+04] Chaki S., Clarke E.M., Groce A., Jha S., Veith H., Modular verification of software components in C. IEEE Transaction in Software Engineering. 30(6), 388–402. Wiley, New York . (2004)

[CGP02] Chandra S., Godefroid P. and Palm C., Software model checking in practice., an industrial case study. In ICSE 02: International Conference on Software Engineering. ACM, 431–441. (2002)

[CR08] Chang B. E. and Rival X., Relational inductive shape analysis. In POPL 08: Principles of Programming Languages. 247–260. (2008)

[CWZ90] Chase D., Wegman M. and Zadeck F., Analysis of pointers and structures. In PLDI90: Programming Languages Design and Implementation. ACM, 296–310. (1990)

[Cim08] Cimatti A., Beyond Boolean SAT: Satisfiability modulo theories. In Discrete Event Systems. WODES08. 9th International Workshop on, pp. 68-73, May (2008)

[CDE08] Cadar C., Dunbar D. and Engler D., KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In 8th USENIX Symposium on Operating Systems Design and Implementation (2008)

[CZB+09] Ciortea L., Zamfir C., Bucur S., Chipounov V., Candea G., Cloud9: A Software Testing Service ACM Operating Systems Review, Vol. 43, No. 4, December (2009)

[CKL04] Clarke E.M., Kroening D., Lerda,F., A tool for checking ANSI-C programs. In Proceeding TACAS, LNCS 2988, Springer pp. 166-176. (2004)

[CGJ+00] Clarke E.M., Grumberg O., Jha S., Lu Y., Veith H., Counterexample-guided abstraction refinement. In: Proc. CAV, LNCS, vol. 1855,pp. 154–169. Springer, Berlin. (2000)

[CE81] Clarke E. M. and Emerson E., Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs. Lecture Notes in Computer Science 131. Springer- Verlag, 52–71. (1981)

[CFJ93] Clarke E. M., Filkorn T. and Jha S., Exploiting symmetry in temporal logic model checking. In CAV 93: Computer Aided Verification. Lecture Notes in Computer Science 697. Springer-Verlag, 450–462. (1993)

[CGP99] Clarke E.M., Grumber O., Peled D., Model Checking. MIT, Cambridge (1999)

[CKS+05] Clarke E.M., Kroening D., Sharygina N., Yorav K., SatAbs: SAT-based predicate abstraction for ANSI-C. In: Proc.TACAS,LNCS, vol. 3440, pp. 570–574. Springer, Berlin. (2005)

[Cla76] Clarke E.M., A system to generate test data and symbolically execute programs. IEEE Transactions in Software Engineering 2(2), 215–222.(1976)

[CC77] Cousot P. and Cousot R., Abstract interpretation: a unified lattice model for the static analysis of programs. In POPL 77: Principles of Programming Languages. ACM, 238–252.(1977)

[CC92] Cousot P., Cousot R., Abstract interpretation frameworks. Journal of Logic and Computation, 2(4):511—547. (1992)

[DLS02] Das M., Lerner S. and Seigle M., ESP: Path-sensitive program verification in polynomial time. In PLDI 02: Programming Language Design and Implementation. ACM, 57–68. (2002)

[DDP99] Das M., Dill D. and Park S. Experience with predicate abstraction. In CAV 99: Computer-Aided Verification. Lecture Notes in Computer Science 1633. Springer-Verlag, 160–171. (1999)

[DLL62] Davis M., . Logemann G. and Loveland L., A machine program for theorem-proving. Communication of the. ACM, vol. 5, no. 7, pp. 394-397. (1962)

[DB08] De Moura L. and Bjørner N., Z3: An efficient SMT solver. In TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems.Lecture Notes in Computer Science 4963. Springer-Verlag, 337–340. (2008)

[DLR06] Deng X., Lee J., and Robby., Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In Automated Software Engineering. (2006)

[Dij76] Dijkstra E. , A Discipline of Programming. Prentice-Hall. (1976)

[Dil96] Dill D., The Murphi verification system. In CAV 96: Computer-Aided Verification. Lecture Notes in Computer Science 1102. Springer-Verlag, 390–393. (1996)

[DOY06] Distefano D., O'Hearn P. and Yang H., A local shape analysis based on separation logic. In TACAS 06: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science 3920. Springer-Verlag, 287–302. (2006)

[DD06a] Dutertre B. and De Moura L., The Yices SMT solver. Tool paper at http: // yices.csl. sri. com/ tool-paper. pdf . (2006)

[DD06b] Dutertre B. and De Moura L., A fast linear-arithmetic solver for DPLL (T). In Computer Aided Verification, pp. 81-94, Springer. (2006)

[DEP+07] Dwyer W., Elbaum S. Person S., and Purandare R., Parallel randomized state-space search. In Software Engineering, 2007. ICSE07. 29th International Conference on, pp. 3-12. (2007)

[ES03] Een N. and Sorensson N., An extensible SAT solver. In SAT 03: Sixth International Conference on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science 2919. Springer-Verlag, 502–518. (2003)

[Eme90] Emerson E., Temporal and modal logic. In Handbook of Theoretical Computer Science, J. van Leeuwen, Ed. Vol. B. Elsevier Science Publishers, 995–1072. (1990)

[ES96] Emerson E. and Sistla A., Symmetry and model checking. Formal Methods in System Design 9, 105–131. (1996)

[FJM05] Fischer J., Jhala R. and Majumdar R,. Joining dataflow with predicates. In ESEC/FSE05: Foundations of Software Engineering. ACM, 227–236. (2005)

[FJL01] Flanagan C., Joshi R. and Leino K. R., Annotation inference for modular checkers. Information Processing Letters 77, 2-4, 97–108. (2001)

[GBW06] Gotlieb A.,Botella B. and Watel M., Inka : Ten years after the first ideas. In 19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06), Paris, France. (2006)

[GDB05] Gotlieb A., Denmat T. and Botella B., Constraint-based test data generation in the presence of stack-directed pointers. In 20th IEEE/ACM International Conference on Automated Software Engineering (ASE05), 7-11. Long Beach, CA, USA, pages 313–316. (2005)

[GHN+04] Ganzinger H., Hagen G., Nieuwenhuis R., Oliveras A., and Tinelli C., DPLL (T): Fast decision procedures. In Computer aided verification, pp. 293-295, Springer, (2004)

[GS05] Grosu S. and Smolka S., Monte Carlo model checking. Lecture notes in computer science, pp. 271-286. (2005)

[God96] Godefroid P., Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. Lecture Notes in Computer Science 1032. Springer- Verlag. (1996)

[God97] Godefroid P., Model checking for programming languages using Verisoft. In POPL97. Principles of Programming Languages. ACM, 174–186. (1997)

[GKS05] Godefroid P., Klarlund N. and Sen K. DART: directed automated random testing. In PLDI 05: Programming Language Design and Implementation. ACM, 213–223. (2005)

[GLM+08] Godefroid P., Levin, Molnar D., et al., Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium, Citeseer, (2008)

[God07] Godefroid P., Compositional Dynamic Test Generation. In POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, (New York, NY, USA), pp. 47-54, ACM (2007)

[GK04] Godefroid P. and Khurshid S., Exploring very large state spaces using genetic algorithms. International Journal on Software Tools for Technology Transfer (STTT), vol. 6, no. 2, pp. 117-127. (2004)

[GH97] Graf S. and Sa¨ıdi H., Construction of abstract state graphs with PVS. In CAV. Lecture Notes in Computer Science 1254. Springer-Verlag, 72–83. (1997)

[GCP99] Grumberg O., Clarke E.M.,and Peled D., Model Checking. MIT Press, (1999)

[GCN+08] Gulavani B., Chakraborty S., Nori A. and Rajamani S. Automatically refining abstract interpretations. In TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science 4963. Springer-Verlag, 443–458. (2008)

[GHK+06] Gulavani B., Henzinger T., Kannan Y., Nori A. and Rajamani S., Synergy: a new algorithm for property checking. In FSE 06: Foundations of Software Engineering. ACM, 117–127. (2006)

[GT07] Gulwani S. and Tiwari A.. Computing procedure summaries for interprocedural analysis. In ESOP, pages 253–267. (2007)

[GT06] Gulwani, S. and Tiwari, A. 2006. Combining abstract interpreters. In PLDI06: Programming Language Design and Implementation. ACM, 376–386. (2006)

[GMR09] Gupta A., Majumdar R. and Rybalchenko A., From tests to proofs. In TACAS 09: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science 5505. Springer-Verlag, 262–276. (2009)

[HA06] Hackett B. and Aiken A., How is aliasing used in systems software? In FSE 06: Foundations of Software Engineering. ACM, 69–80. (2006)

[HP00] Havelund K. and Pressburger T., Model checking Java programs using Java Pathfinder. Software Tools for Technology Transfer (STTT) 2(4), 72–84. (2000)

[HJM+04] Henzinger T., Jhala R., Majumdar R. and McMillan K., Abstractions from proofs. In POPL 04: Principles of Programming Languages. ACM, 232–244. (2004)

[HJM+03a] Henzinger, T., Jhala, R., Majumdar, R., and Qadeer, S. Thread-modular abstraction refinement. In CAV 03: Computer-Aided Verification. Lecture Notes in Computer Science. Springer-Verlag. (2003)

[HJM+03b] Henzinger T.A., Jhala R., Majumdar R., Sanvido M.A.A.: Extreme model checking. In: International Symposium on Verification: Theory and Practice, LNCS, vol. 2772, pp. 332–358. Springer, Berlin. (2003)

[HJM+02] Henzinger T., Jhala R., Majumdar R., and Sutre G. Lazy abstraction. In POPL 02: Principles of Programming Languages. ACM, 58–70. (2002)

[HQR98] Henzinger T., Qadeer S. and Rajamani S. You assume, we guarantee: methodology and case studies. In CAV 98: Computer-aided Verification, A. Hu and M. Vardi, Eds. Lecture Notes in Computer Science 1427. Springer-Verlag, 440–451. (1998)

[HJM04] Henzinger T. A., Jhala R. and Majumdar R. Race checking by context inference. In PLDI04: Programming Languages Design and Implementation. ACM, 1–12. (2004)

[Hin01] Hind M. Pointer analysis: haven't we solved this problem yet? In PASTE. 54–61. (2001)

[Hoa69] Hoare C. An axiomatic basis for computer programming. Communications of the ACM 12, 576–580. (1969)

[Hol75] Holland J. Adaptation in natural and artificial systems. Ann Arbor: The University of Michigan Press, (1975)

[Hol97] Holzmann G. The Spin model checker. IEEE Transactions on Software Engineering 23, 5 (May), 279–295. (1997)

[HFG+94] Hutchins M. H., Foster T., Goradia T., and Ostrand, D. Experiments of the effectiveness of dataflow and control flow based test adequacy criteria," in International Conference on Software Engineering, pp. 191–200. (1994)

[ID96] Ip C. and Dill D. Better verification through symmetry. Formal Methods in System Design 9, 41–75. (1996)

[IYG+08] Ivancic F., Yang Z., Ganai M. K., Gupta A. and Ashar P. Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404, 3, 256–274. (2008)

[IYG+05] Ivancic F., Yang Z., Ganai M. K., Gupta A., Shlyakhter I. and Ashar, P.F-soft: Software verification platform. In CAV 05: Computer-Aided Verification. 301–306. (2005)

[JIG+06] Jain H., Ivancˇic´ F., Gupta A., Shlyakhter I., and Wang C. Using statically computed invariants inside the predicate abstraction and refinement loop. In CAV 06: Computer-Aided Verification. 137–151. (2006)

[JHG+09] Jayaraman K., Harvison D., Ganesh V. and Kiezun A., jFuzz: A concolic whitebox fuzzer for Java. in NASA Formal Methods Symposium. (2009)

[Joh09] Johnson,I. Formal Verification with SMT Solvers: Why and How. (2009)

[Joh02a] Johnson C.G. Deriving genetic programming fitness properties by static analysis. In Proceedings of the 4th European Conference on Genetic Programming (EuroGP02), pages 299–308. Springer, (2002)

[Joh02b] Johnson C.G. Genetic programming with guaranteed constraints. In Recent Advances in Soft Computing, pages 134–140. The Nottingham Trent University, (2002)

[Joh02c] Johnson C.G. What can automatic programming learn from theoretical computer science. In Proceedings of the UK Workshop on Computational Intelligence. University of Birmingham, (2002)

[Joh07] Johnson C.G. Genetic programming with fitness based on model checking. In Proceedings of the 10th European Conference on Genetic Programming (EuroGP07), pages 114–124. Springer, (2007)

[Joh09] Johnson C.G. Genetic programming crossover: Does it cross over? In Proceedings of the 12th European Conference on Genetic Programming (EuroGP09), pages 97–108. Springer, (2009)

[JN95] Jones N.D., and Nielson F. Abstract interpretation: a semantics based tool for program analysis. Handbook of Logic in Computer Science, (1995)

[Jon83] Jones C.Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619. (1983)

[KP08a] Katz G., and Peled D. Model checking-based genetic programming with an application to mutual exclusion. Tools and Algorithms for the Construction and Analysis of Systems, 4963:141–156, (2008)

[KP08b] Katz G., and Peled D. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, 5311:33–47, (2008)

[KP92] Katz S. and Peled D. Verification of distributed programs using representative interleaving sequences. Distributed Computing 6, 2, 107–120. (1992)

[Kei03] Keijzer M. Improving symbolic regression with interval arithmetic and linear scaling. In Proceedings of EuroGP'03, pages 70–82. Springer-Verlag, (2003)

[KPV03] Khurshid S., Pasareanu C. and Visser W. Generalized symbolic execution for model checking and testing. In TACAS'03, (2003)

[KK09] Kim M. and Kim Y. Concolic testing of the multi-sector read operation for flash memory file system. In Brazilian Symposium on Formal Methods, (2009)

[KK11] Kim Y. and Kim M. SCORE: a scalable concolic testing tool for reliable embedded software," in European Software Engineering Conference/Foundations of Software Engineering, Szeged, Hungary, September 5-9, pp. 420–423, tool demonstration track. (2011)

[KKR11] Kim M. and Kim Y. and Rothermel G. Distributed concolic algorithm of the SCORE framework. KAIST, Tech. Rep., (2011)

[Kin76] King J.C.Symbolic execution and program testing. Communications of the ACM, 19(7):385–394. (1976)

[Kin09] King A. Distributed parallel symbolic execution," Kansas State University, Tech. Rep., (2009)

[Knu75] Knuth,D. Estimating the efficiency of backtrack programs. Mathematics of Computation, vol. 29, no. 129, pp. 121-136, (1975)

[Kor90] Korel B. Automated Software Test Data Generation. IEEE Transaction on Software Engineering vol. 16, no. 8, pp. 870-879, (1990)

[Kor96] Korel B. Automated test data generation for programs with procedures. In Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis, pp. 209-215, ACM New York, NY, USA, (1996)

[KCY03] Kroening D., Clarke E. and Yorav K. Behavioral consistency of C and Verilog programs using bounded model checking. In DAC 03: Design Automation Conference. ACM, 368–371. (2003)

[KGC04] Kroening D., Groce A., Clarke E.M., Counterexemple guided abstraction reffinement via program execution. In: Proc. CFEM, LNCS, vol. 3308, pp. 224–238.Springer, Berlin. (2004)

[Lam83] Lamport L. Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems 5, 2, 190–222. (1983)

[LA04] Lattner C. and Adve V.LLVM: A compilation framework for lifelong program analysis & transformation. In Intl. Symp. on Code Generation and Optimization, (2004)

[LY92] Lee D. and Yannakakis M. Online minimization of transition systems. In Proceedings of the 24th Annual Symposium on Theory of Computing. ACM Press, 264–274. (1992)

[LS00] Lev-Ami T. and Sagiv S.TVLA: A system for implementing static analyses. In SAS. Lecture Notes in Computer Science 1824. Springer-Verlag, 280–301. (2000)

[MBC+07] Magill S. Berdine J., Clarke E. M. and Cook B. Arithmetic strengthening for shape analysis. In SAS. 419–436. (2007)

[MS07] Majumdar R. and Sen K., Hybrid concolic testing. In Proceedings of the 29th international conference on Software Engineering, pp. 416-426, IEEE Computer Society, (2007)

[MX07] Majumdar R. and Xu M. Directed test generation using symbolic grammars. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 134-143, ACM New York, NY, USA, (2007)

[MS99] Marques-Silva J. and Sakallah K. GRASP: A search algorithm for propositional Satisfiability. IEEE Transactions on Computers, vol. 48, no. 5, pp. 506-521, (1999)

[MA05] Marinov B., and Agha G. CUTE: A concolic unit testing engine for C. In European Software Engineering Conference Foundations of Software Engineering, (2005)

[McM93] McMillan K. Symbolic Model Checking: An Approach to the State-Explosion Problem Kluwer Academic Publishers. (1993)

[MFR06] Meyer R., Faber J. and Rybalchenko A. Model checking duration calculus: A practical approach. In ICTAC 06: Theoretical Aspects of Computing - Third International Colloquium. Lecture Notes in Computer Science, vol. 4281. Springer-Verlag, 332–346. (2006)

[MSM+08] Mansouri M., Samani, P. Mehlitz C., Pasareanu C., Markosian L., Penix J., Brat G.P., Visser W. Program Model Checking: A Practitioner's Guide. National Aeronautics and Space Administration (NASA) Ames Research Center March . (2008)

[Mau04] Mauborgne .L., Astrée: verification of absence of run-time error. In Building the Information Society, R. Jacquard (Ed.), Kluwer Academic Publishers, pp. 385—392, (2004)

[MMZ+01] Moskewicz M., Madigan C., Zhao Y., Zhang L. and Malik S. Chaff: Engineering an efficient SAT solver. In DAC 01: Design Automation Conference. 530–535. (2001)

[Mye79] Myers G.J. Art of Software Testing. John Wiley & Sons, Inc., New York, (1979)

[NMR+02] Necula G., McPeak, S., Rahul P. and Weimer W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in CC '02: Proceedings of the 11th International Conference on Compiler Construction, (London, UK), pp. 213-228,Springer-Verlag, (2002)

[Nel81] Nelson G. Techniques for program verification. Tech. Rep. CSL81-10, Xerox Palo AltoResearch Center. (1981)

[NO80] Nelson G. and Oppen D. Fast decision procedures based on congruence closure. Journal of the ACM 27, 2, 356–364. (1980)

[OMA+04] Oh y., Mneimneh M., Andraus Z., Sakallah K. and Markov I. AMUSE: A Minimally- Unsatisfiable Subformula Extractor. In Proceedings of the 41st annual Design Automation Conference, p. 523, ACM, (2004)

[PE05] Pacheco C. and Ernst M. Eclat: Automatic generation and classification of test inputs. In 19th European Conference Object-Oriented Programming, pp. 504-527, (2005)

[PV09] Pasareanu C. and Visser W. A survey of new trends in symbolic execution for software testing and analysis. Software Tools for Technology Transfer, vol. 11, no. 4, pp. 339–353, (2009)

[PMB+08] Pasareanu C. Mehlitz P.,Bushnell D., Gundy-burlet K., Lowry M., Person S. and Pape M. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In International Symposium on Software Testing and Analysis, (2008)

[PGB+08] Pasareanu C., Giannakopoulou D., Bobaru M., Cobleigh J. and Barringer H. Learning to divide and conquer: applying the algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32, 3, 175–205. (2008)

[PR10] Păsăreanu C. and Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode. ASE '10 Proceedings of the IEEE/ACM international conference on Automated software engineering Pages 179-180 ACM NY, USA, (2010)

[PR07] Podelski A. and Rybalchenko A. Armc: The logical choice for software model checking with abstraction refinement. In PADL 07: Practical Aspects of Declarative Programming. Lecture Notes in Computer Science 4354. Springer-Verlag, 245–259. (2007)

[QS81] Queille J. and Sifakis J. Specification and verification of concurrent systems in CESAR.In Fifth International Symposium on Programming, M. Dezani-Ciancaglini and U. Montanari, Eds. Lecture Notes in Computer Science 137. Springer-Verlag, 337–351. (1981)

[RHS95] Reps T., Horwitz S. and Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In POPL 95: Principles of Programming Languages. ACM, 49–61. (1995)

[Rey02] Reynolds J. C. Separation logic: A logic for shared mutable data structures. In LICS. 55–74. (2002)

[Sai00] Saidi H. Model checking guided abstraction and analysis. In SAS 00: Static-Analysis Symposium. Lecture Notes in Computer Science 1824, Springer-Verlag, 377–396. (2000)

[SMA05] Sen K., Marinov D. and Agha G.CUTE: a concolic unit testing engine for C. In ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, (New York, NY, USA), pp. 263-272, ACM, (2005)

[SA06a] Sen K. and Agha G. Automated systematic testing of open distributed programs. Lecture notes in computer science, pp. 339-356, (2006)

[SA06b] Sen K. and Agha G. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In Computer Aided Verification, (2006)

[SP81] Sharir M. and Pnueli A. Two approaches to interprocedural data dalow analysis. In Program Flow Analysis: Theory and Applications. Prentice-Hall, 189–233. (1981)

[Sho84] Shostak R.Deciding combinations of theories. Journal of the ACM 31, 1, 1–12. (1984)

[Sho79] Shostak R. A practical decision procedure for arithmetic with function symbols. Journal of the ACM (JACM), vol. 26, no. 2, pp. 351-360, (1979)

[SK10] Siddiqui j and Khurshid S. ParSym: Parallel Symbolic Execution. In International Conference on Software Technology and Engineering, (2010)

[SS96] Silva J. and Sakallah K. A. Grasp - a new search algorithm for satisfiability. In ICCAD 96: International Conference on Computer-Aided Design. ACM, 220–227. (1996)

[SGE00] Sistla A., Gyuris V. and Emerson E.SMC: A symmetry-based model checker for verification of safety and liveness properties. ACM Transactions on Software Engineering Methodology 9, 133–166. (2000)

[Sta85] Stark E.A proof technique for rely/guarantee properties. In FSTTCS 85: Foundations of Software Technology and Theoretical Computer Science, S. N. Maheshwari, Ed. Lecture Notes in Computer Science 206. Springer-Verlag, 369–391. (1985)

[SP10] Staats M. and Pasareanu C. Parallel symbolic execution for structural test generation. In International Symposium on Software Testing and Analysis, (2010)

[Ste96] Steensgard B.Points-to analysis in almost linear time. In POPL 96: Principles of Programming Languages. ACM, 32–41. (1996)

[TdH08] Tillmann N. and de Halleux J. Pex-white box test generation for.NET. In Bernhard Beckert and Reiner Hähnle, editors, Tests and Proofs, Second International Conference, TAP08, Prato, Italy, April 9-11, Proceedings, volume 4966 of Lecture Notes in Computer Science, pages 134–153. Springer, (2008)

[TS06] Tillmann N. and Schulte W. Unit tests reloaded : Parameterized unit testing with symbolic execution. Technical Report MSR-TR-2005-153, Microsoft Research (MSR), March (2006)

[TS05] Tillmann N. and Schulte W. Parameterized unit tests. In European Software engineering Conference/Foundations of Software Engineering, (2005)

[TBV07] Tomb,A., Brat G. and Visser W.Variably interprocedural program analysis for runtime error detection. In Proceedings of the international symposium on Software testing and analysis, p. 107, ACM, (2007)

[Val92] Valmari A. A stubborn attack on state explosion. Formal Methods in System Design 1, 497–322. (1996)

[VW84] Vardi M. and Wolper P. Reasoning about infinite computations. Information and Computation 115, 1, 1–37. (1994)

[VHB+03] Visser W. Havelund K. Brat G., Park S. and Lerda F. Model checking programs. Automated Software Engineering Journal. (2003)

[VHB+00] Visser W., Havelund K., Brat G. and Park S. Model checking programs. In Automated Software Engineering, (2000)

[WMMR05] Williams N., Marre B., Mouy P.and Roger M. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings, volume 3463 of Lecture Notes in Computer Science, pages 281–292. Springer, (2005)

[WYG+07] Wang C., Yang Z., Gupta A. and Ivancic F. Using counterexamples for improving the precision of reachability computation with polyhedra. In CAV 07: Computer-Aided Verification.352–365. (2007)

[WL04] Whaley J. and Lam M. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI 04: Programming Language Design and Implementation.131–144. (2004)

[XVM05] Xia S., Vito B. and Mun˜oz C. Automated test generation for engineering applications. In ASE 05: Automated Software Engineering. ACM, 283–286. (2005)

[XA05] Xie Y. and Aiken A.Scalable error detection using Boolean satisfiability. In POPL 05: Principles of Programming Languages. ACM. (2005)

[Yah01] Yahav E. Verifying safety properties of concurrent Java programs using 3-valued logic. In POPL 01: Principles of Programming Languages. ACM Press, 27–40. (2001)

[YLB+08] Yang H., Lee O., Berdine J., Calcagno C., Cook B., Distefano D. and O'Hearn P. Scalable shape analysis for systems code. In CAV 08: Computer-Aided Verification. Lecture Notes in Computer Science 5123. Springer-Verlag, 385–398. (2008)

[YWG+06] Yang Z., Wang C., Gupta A. and Ivancic, F. Mixed symbolic representations for model checking software programs. In MEMOCODE. 17–26. (2006)

[Yan90] Yannakakis M. Graph theoretic methods in database theory. In Proc. 9th ACM Symp. on Principles of Database Systems. ACM Press, 203–242. (1990)

[YP05] Young M., Pezze M. Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2005)

[ZMM+01] Zhang L., Madigan C., Moskewicz M., and Malik S. Efficient conflict driven learning in a boolean satisfiability solver. In Proceedings of the IEEE/ACM international conference on Computer-aided design, p. 285, IEEE Press (2001)