HAL
open science

Contributions à la génération de tests à base de contraintes

Arnaud Gotlieb

To cite this version:

Arnaud Gotlieb. Contributions à la génération de tests à base de contraintes. Génie logiciel [cs.SE]. Université Européenne de Bretagne, 2011. tel-00699260

HAL Id: tel-00699260

https://theses.hal.science/tel-00699260

Submitted on 20 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Année 2011

Habilitation à diriger des recherches / UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

Mention : Informatique

Présentée par

Arnaud Gotlieb

préparée à l'unité de recherche UMR 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Composante Universitaire : ISTIC

Contributions à la génération de tests à base de contraintes

Habilitation soutenue à Rennes
le 12 Décembre 2011

devant le jury composé de :

Claude Jard
Professeur à ENS Rennes, France / Président

Yves Ledru
Professeur à l'Université de Grenoble, France / Rapporteur

Bruno Legeard
Professeur à l'Université de Besancon, France / Rapporteur

Pascal Van Hentenryck
Professor at Brown University, Providence, USA / Rapporteur

Patrice Godefroid
Principal Research Scientist at Microsoft Research, Richmond, USA / Examinateur

Thomas Genet
Professeur à l'Université de Rennes / Examinateur

Thomas Jensen
Directeur de Recherche à l'INRIA / Examinateur

Remerciements

Contents

Remerciements **3**

Introduction **7**
0.1 Contexte 7
0.2 Chronologie des contributions 10
0.3 Organisation du mémoire 13

I Fondements **15**

1 Les origines **17**
Automatic test data generation using constraint solving techniques
A. Gotlieb, B. Botella, and M. Rueher. 18

2 Test logiciel à base de contraintes **25**
A CLP framework for computing structural test data
A. Gotlieb, B. Botella, and M. Rueher 26

3 Contraintes et abstractions **37**
EUCLIDE: A constraint-based testing platform for critical c programs
A. Gotlieb 40
An abstract interpretation based combinator for modeling while loops in constraint programming
T. Denmat, A. Gotlieb, and M. Ducasse 46
Constraint solving on modular integers
A. Gotlieb, M. Leconte, and B. Marre 55

II Développements **65**

4 Oracles **67**
Exploiting symmetries to test programs
A. Gotlieb 68

5 Modélisation à contraintes des programmes avec pointeurs **75**
Goal-oriented test data generation for pointer programs
A. Gotlieb, T. Denmat, and B. Botella 76
Modelling dynamic memory management in constraint-based testing
F. Charreteur, B. Botella, and A. Gotlieb 91

6 Modélisation à contraintes des constructions orientées-objet **105**
Constraint-based test input generation for java bytecode
F. Charreteur and A. Gotlieb 106

7 Modélisation à contraintes des calculs flottants **113**
Symbolic execution of floating-point computations
B. Botella, A. Gotlieb, and C. Michel 115

III Applications **139**

8 Génération de tests pour Java Card **141**
Using chrs to generate test cases for the JCVM
S.D. Gouraud and A. Gotlieb 142
A semi-empirical model of test quality in Symmetric Testing: Application to testing Java Card APIs
A. Gotlieb and P. Bernard 151

9 Système d'alerte et anti-collision (TCAS) **157**
TCAS software verification using constraint programming
A. Gotlieb 157

IV Bilan et Perspectives **169**

10 Bilan **171**

11 Perspectives **173**

V Annexe : Curriculum Vitae **175**

Bibliographie **205**

Introduction

0.1 Contexte

En termes de fondements, de développements et d'applications, la Programmation par Contraintes et le Test Logiciel sont deux domaines scientifiques qui ont a priori peu de choses en commun. En bref, la *Programmation par Contraintes* est un paradigme permettant la résolution de problèmes combinatoires difficiles issus par exemple d'applications de planification ou d'ordonnancement de tâches, tandis que le *Test Logiciel* est un ensemble de techniques, méthodes et processus visant à évaluer la correction des programmes informatiques. Et pourtant, depuis une vingtaine d'années maintenant, plusieurs ponts solides ont été édifiés entre ces deux domaines. La terminologie "Test à Base de Contraintes" a ainsi été forgée pour décrire ce nouveau champ de recherche et d'applications. Ce mémoire tente de faire une première synthèse sur les avancées en *Test à Base de Contraintes (CBT)*, en mettant en exergue nos contributions dans ce domaine. Nous démarrons notre cheminement au travers ce champ de recherche par une introduction des approches existantes pour la vérification des logiciels critiques, qui constituent la motivation principale de notre travail.

Dans les systèmes critiques, les logiciels sont considérés comme le maillon faible de la chaîne. En effet, ceux-ci sont souvent développés selon des méthodes artisanales qui reposent plus sur le savoir-faire et l'expertise des ingénieurs, que sur des méthodologies de développement garantissant la production de logiciels sûrs. Malgré cela, les logiciels produits sont de grande qualité et généralement très sûrs, mais aussi vulnérables car leur processus de développement n'est pas insensible aux erreurs humaines. De plus, la complexité et la taille des logiciels critiques, en particulier dans le domaine de l'embarqué, n'ont cessé de croître ces dernières années, ce qui impose dorénavant la mise au point de techniques automatisées pour leur vérification.

Plusieurs techniques, reposant sur des fondements théoriques solides, ont été proposés pour la vérification des logiciels critiques. On distingue généralement les techniques de preuve, qu'elles soient basées sur l'utilisation d'assistants interactifs ou de démonstrateurs automatiques, les techniques de "model-checking" et d'analyse statique, et les techniques de test logiciel. Ce qui distingue ces dernières par rapport aux autres, est qu'elles réclament l'exécution du programme à vérifier, ce qui leurs confèrent plusieurs avantages indiscutables. Tout d'abord, le test

permet la vérification du code binaire réellement exécuté et non pas seulement du code source. Ainsi, c'est bien le programme utilisé de manière opérationnelle qui est vérifié et non pas une version originale écrite dans un langage de plus haut-niveau, qui sera transformée par un compilateur plus ou moins optimisant. Ensuite, la vérification du logiciel par le test s'opère sans hypothèse sur la sémantique du langage de programmation utilisé ou sur le modèle d'exécution du programme. Ces hypothèses sont parfois hasardeuses dans le contexte des logiciels embarqués pour lesquels les cibles d'exécution possèdent certaines spécificités[1]. Enfin, le test permet la vérification (partielle) du logiciel dans son contexte d'utilisation et son environnement d'exécution. Le logiciel est testé en utilisant les librairies réellement liées, dans un processus géré par le système d'exploitation de la machine cible qui peut éventuellement ne disposer que de ressources limitées. Ces conditions font du test le moyen actuellement privilégié pour vérifier les logiciels dans le monde industriel. Au chapitre des désavantages, la limitation intrinsèque de cette forme de vérification est son incomplétude. Ainsi, le grand Edsger W. Dijkstra écrivait: "Program testing can be used to show the presence of bugs, but never to show their absence!". En effet, à moins d'exécuter le programme sur tout son domaine d'entrée et de vérifier toutes les sorties calculées, le test ne peut offrir de garantie absolue sur la correction d'un programme sous test, ou sur l'absence de certaines fautes puisque de nombreux comportements du programme risquent de ne pas être évalués. Bien qu'elle soit souvent utilisée pour dénigrer le Test Logiciel, cette limitation intrinsèque est relativement bien acceptée en pratique car d'une part la qualité des tests est grande et d'autre part, les logiciels à vérifier ont des modèles de fiabilité qui tolèrent les fautes, dès lors qu'elles ne sont pas trop fréquentes[2]. En fait, les difficultés du test logiciel proviennent plus du coût de sa mise en oeuvre que de cette limitation théorique. En effet, chaque test nécessite une définition précise de son objectif, c'est à dire des raisons qui ont poussées à sa sélection, l'écriture et la validation d'un script de test, ainsi que la prédiction des résultats attendus du programme. Ce dernier point est particulièrement délicat puisqu'il nécessite de la part des ingénieurs, une connaissance approfondie de l'application sous test. Ces éléments sont coûteux à développer, à maintenir et à documenter car ils rentrent également dans le processus de développement logiciel, au même titre que les spécifications et le code source. C'est la raison pour laquelle un enjeu particulièrement important dans le domaine du Test Logiciel, concerne l'automatisation de la production des tests. Dans l'industrie du logiciel embarqué, cet enjeu est de taille puisqu'on estime que la phase de validation du logiciel représente au moins la moitié du coût de son développement.

Afin de répondre à cet enjeu, les chercheurs se sont intéressés aux deux grands problèmes fondamentaux suivants :

[1]Par exemple, la correction d'un programme qui manipule des nombres à virgule flottante ne peut être établie sans une connaissance approfondie de la cible d'exécution (format des registres du processeur, mode d'arrondi des calculs intermédiaires, etc.)

[2]Par exemple, il est accepté que même un système ultra-critique tel que le système de commandes de vol d'un avion de ligne puisse défaillir une fois (i.e., mener à un évènement catastrophique) pendant 10^9 heures d'utilisation [Littlewood 93]. Notons tout de même que pour une flotte de 1000 avions sur une durée de vie de 30 ans, cela correspond à une probabilité de 0.1 d'observer une telle défaillance

- **le problème de la sélection des données de test**. Il s'agit ici d'identifier dans une espace de Recherche de taille non bornée ou très grand, les données d'entrée à utiliser pour évaluer la correction du programme sous test. Cette sélection vise à choisir les données les plus susceptibles de détecter des fautes dans les programmes, bien que nous n'ayons aucune connaissance de celles-ci. En pratique, cette sélection est faite soit à partir d'un modèle issu des spécifications (test à base de modèles, test fonctionnel), soit à partir du programme sous test lui-même (test à partir de code, test structurel), soit à partir de modèles de fautes (test mutationnel), ou encore à partir d'un modèle d'usage du logiciel (test statistique). Dans la plupart des cas, la sélection de données d'entrée repose sur le choix de critères de test qui permettent de rationaliser le processus de test. Ces critères de test permettent également de limiter la taille des jeux de test. Le point d'achoppement de ces techniques concerne l'établissement d'un niveau de confiance suffisant dans la couverture des comportements du programme et résulte d'un compromis entre le coût du test et le niveau de confiance attendu. Ce problème de la sélection apparaît également lorsque des tests existants sont rejoués pour des versions antérieures du programme (tests de non-régression) et la question de l'ordre dans lequel sont exécutés les cas de test est alors crucial pour démasquer des fautes le plus rapidement possible.

- **le problème de l'oracle**. L'exécution du programme avec les données de test produit des sorties qui doivent être contrôlées avec une procédure manuelle ou automatique, l'oracle de test. L'obtention d'un oracle correct et complet, c'est à dire sans erreur et capable de répondre pour n'importe quelle donnée de test est illusoire de par la complexité des logiciels modernes et des compromis acceptables doivent être trouvés. En pratique, c'est la connaissance approfondie du logiciel sous test qui permet l'écriture d'oracles et l'enjeu des méthodes de test modernes consistent à réussir à produire automatiquement ces oracles à partir de modèles ou d'autres programmes exécutables.

De manière un peu inattendue, certaines instances de ces deux problèmes sont parfois hautement combinatoires. D'une part, l'espace de recherche constitué par les domaines d'entrée et de sortie des programmes peut-être immense, voire infini, et d'autre part, les objectifs à atteindre que l'on dérive des critères de test peuvent caractériser une portion très réduite de cet espace. Par exemple, sélectionner une donnée de test qui atteint un point très imbriqué dans un programme itératif de tri ou bien sélectionner un comportement de programme qui provoque un débordement de capacité mémoire reviennent tous deux à rechercher des aiguilles dans une botte de foin. Par extension, générer automatiquement un jeu de test qui couvre un critère de test ou bien sélectionner des comportements sur un modèle afin d'atteindre une cible de test sont aussi des questions qui mettent en évidence une explosion combinatoire, de par le nombre de chemins d'exécution possibles, ou bien le nombre de comportements possibles du modèle. La notion de "Test à Base de Contraintes" a ainsi été introduite pour couvrir différentes applications qui incluent la génération de données de test structurel [Gotlieb 00b,

Meudec 01, Sy 03, Denmat 07b, Boonstoppel 08, Gotlieb 09a, Charreteur 09, Degrave 09, Charreteur 10a], la génération de cas de test fonctionnels pour microprocesseurs [Lewin 95, Bin 02, Hari 08], la génération de test à partir de modèles [Carver 96, Jackson 00, Legeard 01, Pretschner 01, Bouquet 05], ou bien la recherche de contre-exemples face à des propriétés de programmes [Gotlieb 03b, Denmat 05, Collavizza 07, Collavizza 08].

Les problèmes mentionnés précédemment ont suscité de nombreux travaux de Recherche et ont été attaqués depuis longtemps avec des outils très différents. Un regard exhaustif sur ces travaux déborderait largement la portée de ce document et nous nous restreindrons volontairement à ceux s'appuyant sur la Programmation par Contraintes [Mackworth 77, Hentenryck 92].

Un des points clef de la Programmation par Contraintes est le remplacement de la notion impérative d'*instruction* par celle, déclarative, de *relation*. Une relation contraint les variables du programme et définit implicitement une portion d'un espace de recherche que des techniques puissantes de recherche peuvent alors explorer. Ainsi, un programme à contraintes ne calcule pas la valeur de sortie d'une fonction issue de la sémantique dénotationnelle d'un programme mais il recherche la ou les solutions d'un système de contraintes qui capture la sémantique relationnelle de ce programme.

Les relations d'un programme à contraintes sont soit natives du langage de programmation sous-jacent, soit définies par l'utilisateur. Dans le premier cas, les relations sont générales et utiles lorsque la modélisation du problème à résoudre ne pose pas vraiment de difficultés. Dans le deuxième cas, les relations sont particulières et peuvent être finement adaptées au problème à résoudre. Cette capacité nous est apparue cruciale pour aborder les problèmes du Test Logiciel mentionnés plus haut ; nous y reviendrons. Ainsi, tout un florilège de contraintes particulières[3] existe pour modéliser divers problèmes combinatoires, ainsi que des outils permettant la définition de nouvelles relations. Un programme à contraintes peut être vu comme un modèle de spécification déclaratif, et surtout exécutable. Les relations spécifient le problème à résoudre tandis que sa résolution est laissée aux résolveurs de contraintes, qui sont adaptés aux différents domaines du calcul (domaines finis, domaines numériques continus, mots, listes, etc.). D'une part, la flexibilité offerte par les contraintes quant à la modélisation de problèmes combinatoires, et d'autre part la disponibilité de résolveurs optimisés et extensibles, constituent les éléments déterminants qui ont conduit à expérimenter leur utilisation dans le contexte du Test Logiciel.

0.2 Chronologie des contributions

Depuis une quinzaine d'années, notre approche vise à explorer l'apport de la Programmation par Contraintes à la génération automatique de test pour les programmes impératifs. Notre thèse est qu'il est possible d'attaquer les problèmes combinatoires de la génération de données de test et de production de l'oracle à

[3] La communauté utilise le terme de *contraintes globales* pour désigner ces relations

l'aide de techniques issues de la Programmation par Contraintes. Pour en faire la démonstration, un modèle à contraintes qui capture la sémantique opérationnelle (sans erreur) du programme impératif original est construit et utilisé dans différentes techniques de génération automatique de tests. Ce modèle est bien entendu réversible, c'est à dire qu'il peut être utilisé pour calculer des sorties du programme impératif en fonction des entrées, mais aussi l'inverse, ou bien encore pour générer des entrées en fonction de contraintes additionnelles précisant des objectifs d'atteignabilité dans le code source. Par exemple, la spécification d'une instruction à atteindre conduit à une requête sur ce modèle, qui lorsqu'elle est résolue par un résolveur de contraintes approprié, permet de générer une donnée de test qui atteint le point sélectionné. Ce modèle à contraintes a été développé et enrichi au cours de ces quinze dernières années et constitue la partie principale de nos contributions.

Notre approche de génération automatique a été développée au travers de la réalisation et l'expérimentation de plusieurs prototypes logiciels. Le modèle à contraintes du logiciel InKa, proposé à la fin des années 90, utilisait la propagation de contraintes sur les domaines finis, le filtrage par consistances partielles et des stratégies de recherche génériques telles que *"first-fail"* ou *"iterative domain-splitting"* [Gotlieb 98]. Même si ce modèle nous a permis d'obtenir des résultats expérimentaux intéressants sur des programmes C issus du domaine de l'avionique militaire [Gotlieb 00b], le sous-ensemble du langage C traité restait assez pauvre. Au début des années 2000, nous nous sommes consacrés à plusieurs extensions de ce modèle. Nous avons abordé le problème de la modélisation des pointeurs et de la synonimie[4], c'est à dire de la possibilité de décrire la même case mémoire à l'aide de différentes expressions syntaxiques. Ces travaux ont donné lieu à un modèle à contraintes capable de générer des tests en présence de pointeurs vers les zones nommées de la mémoire [Gotlieb 05a, Gotlieb 07] et un modèle pour la gestion des structures de données dynamiques [Charreteur 09]. Ces modèles ont été implantés et expérimentés dans une nouvelle version du logiciel InKa [Gotlieb 06b], qui est considéré comme un outil pionnier dans le domaine du *Test à Base de Contraintes* [Bardin 09].

En parallèle, nous nous sommes intéressés au problème de l'oracle en test logiciel dont nous avons parlé plus haut. En effet, aucune technique de génération automatique de cas de test ne peut être pleinement opérationnelle si elle ne s'accompagne d'un procédé pour contrôler les sorties calculées. Nous avons proposé une définition générale de la symétrie dans les propriétés de programmes impératifs [Gotlieb 03a] et suggéré son utilisation en tant qu'oracle de tests dans un cadre applicatif intéressant [Gotlieb 06a]. Une approche complémentaire a également été explorée pour l'oracle, au travers de l'utilisation des "Constraint Handling Rules" (CHRs) pour la génération automatique de tests [Gouraud 06]. Les relations de symétries dans les programmes impératifs sont une forme particulière de *relations métamorphiques* et nous avons proposé d'utiliser les contraintes pour produire automatiquement,

[4] "Pointer aliasing" en Anglais

lorsqu'elles existent, des données de test qui invalident ces relations [Gotlieb 03b].

Avec la thèse de Matthieu Petit [Petit 08], nous nous sommes intéressés à une version probabiliste de ce modèle à contraintes. Nous avons proposé d'une part, des opérateurs à contraintes qui modélisent des choix probabilistes partiellement connus [Petit 07a], et l'utilisation de ces opérateurs pour générer automatiquement des tests statistiques structurels [Petit 07b]. Cette forme de test logiciel nécessite la sélection uniforme de chemins faisables dans un graphe de flot de contrôle. Ces travaux nous ont conduit d'une part, à une extension théorique des critères de test pour la prise en compte de chemins non terminant [Gotlieb 09c] et d'autre part, à proposer une nouvelle technique de génération de test aléatoire où chaque élément du sous-domaine d'entrée associé à un chemin a la même probabilité d'être choisi ("Path-oriented Random Testing") [Gotlieb 08, Gotlieb 10b].

Dans le contexte de l'interaction entre méthodes de test logiciel et d'analyse statique, la thèse de Tristan Denmat [Denmat 08], co-encadrée avec Mireille Ducassé, a permis d'étudier les apports de techniques d'Interprétation Abstraite pour la résolution de contraintes et au test logiciel. Nous avons ainsi défini une méthode de résolution de contraintes non-linéaires (i.e., disjonctives, multiplicatives, etc.) sur les domaines finis, qui combine finement le domaine abstrait des polyèdres et le filtrage par consistance de bornes [Denmat 07b]. Puis, nous avons proposé un opérateur à contraintes qui modélise un calcul de boucle, et implémente des règles de déduction basées sur l'élargissement[5] sur les intervalles et les polyèdres [Denmat 07a].

Pour l'exécution symbolique de calculs sur les nombres flottants, nous avons proposé FPSE, un solveur de contraintes arithmétiques sur les flottants [Botella 06]. Ce solveur implémente une consistance de bornes correcte en présence des quatre opérations arithmétiques de base. Tout récemment, nous avons étendu FPSE avec une propriété sur la représentation des nombres flottants, ce qui nous a permis d'obtenir des premiers résultats en matière de génération automatique de données de test [Carlier 11b].

L'exploitation des polyèdres et des relaxations linéaires de contraintes nous a permis de bâtir un nouveau modèle à contraintes qui est à la base de l'outil EUCLIDE [Gotlieb 09a]. Cet outil a été expérimenté dans le contexte de la modélisation de propriétés d'anticollision vol issus de l'avionique civile [Gotlieb 09b]. Dans le cadre de cette expérience, nous avons également proposé des opérateurs de filtrage pour le traitement des contraintes sur les entiers modulaires [Gotlieb 10a].

La thèse de Florence Charreteur [Charreteur 10b] nous a permis d'étendre le modèle à contraintes pour des programmes en Bytecode Java [Charreteur 10a]. Le traitement d'un langage à objet avec des contraintes nous a conduit à étendre le cadre classique des solveurs de contraintes ensemblistes vers des opérateurs sur

[5] "Widening techniques" en Anglais

les ensembles non bornés [Charreteur 08]. Notre modèle à contraintes ensemblistes, implanté dans l'outil JAUT, est suffisamment riche pour traiter des extensions de la Programmation orientée objet telles que l'héritage et le polymorphisme par invocation de méthodes virtuelles [Charreteur 10a].

0.3 Organisation du mémoire

Ce mémoire d'habilitation vise à mettre en exergue nos contributions dans le domaine du *test à base de contraintes*. Nous avons fait le choix de présenter ce mémoire sous forme d'une sélection de nos articles principaux, accompagnée d'un commentaire sur le contexte et la portée de chacun d'entre eux. Notre souhait est de guider le lecteur au travers ce champ de recherche en l'invitant à lire certains articles, mais sans lui imposer. Nos contributions sont tournées vers deux domaines scientifiques distincts : le test logiciel et la programmation par contraintes ; nous avons essayé de respecter cette parité dans le mémoire.

Les choix qui ont présidés à la sélection des articles commentés de ce mémoire ont été de nature diverse, guidés par les principes suivants. Premièrement, ces articles contiennent à nos yeux la description technique la plus fidèle de nos contributions. Souvent, plusieurs tentatives sont nécessaires pour aboutir à ce type de description. Nous avons fait le choix, non pas de la publication la plus visible, mais plutôt de celle qui présente le plus fidèlement une idée, un développement théorique ou un prototype de recherche. Deuxièmement, de par leur variété et leur pluralité, les articles choisis illustrent selon nous les différentes facettes de notre domaine de recherche. En d'autres termes, nous avons visé ici la variété, plutôt que l'exhaustivité. Enfin, nous n'avons choisi que des articles dans lesquels notre participation a été importante en termes de contribution et de rédaction. Ainsi, nous avons sélectionné treize articles, cinq ayant été publiés côté "contraintes" et huit ayant été publiés côté "test". Un Curriculum Vitae contenant l'ensemble de nos publications est donné en Annexe.

Les articles abordent le *test à base de contraintes*, selon l'épine dorsale fondements-développements-applications qui architecture ce mémoire. Nos travaux de recherche se sont portés sur ces trois piliers et il nous a semblé judicieux d'illustrer chacun d'entre eux. Pour chaque article, nous avons donné un commentaire plus ou moins bref afin de replacer l'article dans son contexte et son état de l'Art, et d'en discuter la portée lorsque cela était justifié.

Le mémoire est organisé en quatre parties distinctes. La première partie est consacrée aux fondements du test à base de contraintes avec un chapitre 1 sur les origines de cette approche, un chapitre 2 qui introduit le premier modèle à contraintes que nous avons proposé pour générer des données de test pour un langage réaliste, et un chapitre 3 qui présente nos travaux sur l'utilisation de techniques de calculs dans les domaines abstraits dans les solveurs de contraintes. Cette idée s'est révélée féconde pour le test à base de contraintes, et plus généralement la vérification de programmes, comme nous l'évoquerons plus loin.

La seconde partie se concentre sur les développements du test à base de con-

traintes. Le chapitre 4, le premier de cette partie, s'intéresse à la problématique de l'oracle et présente les notions de test symétrique et de relations métamorphiques. Le chapitre 5 introduit nos développements en matière de modélisation des pointeurs et de traitement de la synonimie. Le chapitre 6 présente nos développements les plus récents en matière de modélisation de l'héritage et d'invocation de méthodes virtuelles, c'est à dire de constructions orientées-objet. Enfin, le chapitre 7 présente nos développements en matière de raisonnement pour les calculs en nombres à virgule flottante. En particulier, ce chapitre détaille notre solveur à contraintes sur les flottants qui est une contribution importante à nos yeux.

La troisième partie est consacrée aux applications du test à base de contraintes, avec un chapitre 8 concernant la génération de test pour la plateforme Java Card et un chapitre 9 concernant la vérification à base de contraintes du Traffic anti-Collision Avoidance System (TCAS).

Enfin, la quatrième partie du mémoire contient un chapitre 10 qui présente un premier bilan de nos travaux dans le domaine du test à base de contraintes et un chapitre 11 qui dresse quelques perspectives.

Part I

Fondements

Chapter 1

Les origines

Contexte

L'idée d'utiliser des contraintes pour automatiser le test des logiciels prend ses racines vers le milieu des années quatre-vingts avec les travaux pionniers de Nicole Choquet et Luc Bougé [Choquet 86,Bougé 86] puis ceux de Bruno Marre [Marre 91], et Dick et Faivre [Dick 93]. Ces auteurs se sont intéressés à la génération automatique de cas de test fonctionnel à partir de spécifications algébriques, en utilisant la Programmation Logique avec Contraintes. A partir d'une spécification formelle de types de données abstraits, des méthodes et outils ont été proposés pour générer automatiquement des jeux de test respectant certaines hypothèses d'uniformité et de régularité. Cette même période a vu également l'aboutissement de nombreux travaux visant à utiliser la Programmation Logique avec Contraintes comme outil de spécification, pour la génération de tests. On peut citer par exemple les travaux de Michael Gorlick et al. [Gorlick 90] et Pesch et al. [Pesch 85] qui soulignaient déjà la pertinence de la réversibilité des contraintes pour la validation de tests existants par les spécifications. On peut également citer ceux de Paul Strooper et Daniel Hoffman [Hoffman 91, Strooper 91] qui contiennent des prémisses sur l'usage de contraintes arithmétiques pour la validation de modules écrits dans un langage de bas niveau tel que C. Au milieu des années quatre-vingt-dix, Bruno Legeard a initié des travaux visant à utiliser un solveur de contraintes ensemblistes pour la génération automatique de cas de test pour des modèles formels B [Legeard 01]. L'idée de base consistait à sélectionner des comportements d'un modèle logico-ensembliste, et à trouver une instanciation des variables d'état de ce modèle permettant d'activer ces comportements. Il est frappant de constater que la plupart de ces travaux initiaux ont connu d'importants développements en Europe dans les années 2000. Les travaux initiaux de Bruno Marre ont conduit au développement industriel de GATEL [Marre 00], un générateur de tests pour les programmes Lustre, au CEA, tandis que les travaux de Bruno Legeard ont conduit à la commercialisation d'un générateur de tests au travers la création de la société *Smartesting*.

Aux Etats-Unis à la fin des années quatre-vingts, Jeff Offutt a proposé une méthode de génération automatique de données de test pour le test de mutation

de programmes Fortran [Offutt 88]. Cette méthode s'appuyait sur une procédure de résolution de systèmes de contraintes relevant implicitement de la *Programmation par Contraintes*. En s'inspirant des idées de Bicevskis *et al.* [Bicevskis 79] développées dix ans auparavant, une méthode de propagation pour les contraintes d'inégalités, qui réduit les domaines de variation de chacune des variables du programme, était proposée pour identifier des données de test capables de "tuer" des "mutants" de programmes [DeMillo 91]. Cette approche était aussi reliée à l'exécution symbolique [King 76, Clarke 76] qui trouvait ici une de ses applications les plus prometteuses. Les techniques de recherche locale ont également été explorées pour la génération automatique de cas de test durant la décennie quatre-vingt-dix avec les travaux de Bogdan Korel [Korel 90], de Neelam Gupta [Gupta 98] et de Nigel Tracey [Tracey 98].

C'est dans ce contexte qu'a été proposé, en 1998, une méthode et un outil pour la génération automatique de données de test structurel pour les programmes C, s'appuyant explicitement sur la Programmation par Contraintes.

A. Gotlieb, B. Botella, and M. Rueher. *Automatic test data generation using constraint solving techniques*. **In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98), pages 53-62, Clearwater Beach, FL, USA, 1998.**

.

Automatic Test Data Generation using Constraint Solving Techniques

Arnaud Gotlieb
Dassault Electronique
55 quai Marcel Dassault
92214 Saint Cloud, France
and also at
Université de Nice - Sophia Antipolis
Arnaud.Gotlieb@dassault-elec.fr

Bernard Botella
Dassault Electronique
55 quai Marcel Dassault
92214 Saint Cloud, France
Bernard.Botella@dassault-elec.fr

Michel Rueher
Université de Nice - Sophia Antipolis
I3S-CNRS Route des colles, BP 145
06903 Sophia Antipolis, France
rueher@unice.fr

Abstract

Automatic test data generation leads to identify input values on which a selected point in a procedure is executed. This paper introduces a new method for this problem based on constraint solving techniques. First, we statically transform a procedure into a constraint system by using well-known "Static Single Assignment" form and control-dependencies. Second, we solve this system to check whether at least one feasible control flow path going through the selected point exists and to generate test data that correspond to one of these paths.

The key point of our approach is to take advantage of current advances in constraint techniques when solving the generated constraint system. Global constraints are used in a preliminary step to detect some of the non feasible paths. Partial consistency techniques are employed to reduce the domains of possible values of the test data. A prototype implementation has been developped on a restricted subset of the C language. Advantages of our approach are illustrated on a non-trivial example.

Keywords

Automatic test data generation, structural testing, constraint solving techniques, global constraints

1 INTRODUCTION

Structural testing techniques are widely used in unit or module testing process of software. Among the structural criteria, both statement and branch coverages are commonly accepted as minimum requirements. One of the difficulties of the testing process is to generate test data meeting these criteria.
From the procedure structure alone, it is only possible to generate input data. The correctness of the output of the execution has to be checked out by an "oracle".

Two different approaches have been proposed for automatic test data generation in this context. The initial one, called *path-oriented* approach [4, 7, 16, 20, 3], includes two steps which are :

- to identify a set of control flow paths that covers all statements (resp. branches) in the procedure ;
- to generate input test data which execute every selected path.

Among all the selected paths, a non-negligeable amount is generally non-feasible [24], i.e. there is no input data for which such paths can be executed. The static identification of non-feasible paths is an undecidable problem in the general case [1]. Thus, a second approach called *goal-oriented* [19] has been proposed. Its two main steps are :

- to identify a set of statements (resp. branches) the covering of which implies covering the criterion ;
- to generate input test data that execute every selected statement (resp. branch).

Assuming that every statement (resp. branch) is reachable, there is at least one feasible control flow path going through the selected statement (resp. branch). The goal of the data generation process is then to identify input data on which one such path is executed.

For these approaches, existing generation methods are based either on symbolic execution [18, 4, 16, 7, 10], or on the so called "dynamic method" [20, 19, 11, 21].

Symbolic execution consists in replacing input parameters by symbolic values and in statically evaluating the statements along a control flow path. The goal of symbolic execution is to identify the constraints (either equalities or inequalities) called "path conditions" on symbolic input values under which a selected path is executed. This method leads to several problems : the growth of intermediate algebraic expressions, the difficulty to deal with arrays (although some solutions exist [13, 8]), and the aliasing problem for pointer analysis. Using symbolic execution corresponds to an exhaustive exploration of all paths going through a selected point. Of course, this may be unacceptable for programs containing a large number of paths.

Korel proposes in [20] to base the test data generation process on actual executions of programs. Its method is called the "dynamic method". If an undesirable path is observed during the execution flow monitoring, then a function minimization technique is used to "correct" the input variables. [19] presents an extension of the dynamic method to the *goal-oriented approach*. This method is designed to handle arrays, dynamic structures, and procedures calls [21]. However, although the dynamic method takes into account some of the problems encountered with symbolic execution, it may require a great number of executions of the program.

This paper introduces a new method to identify automatically test data on which a selected point in the procedure is executed. The proposed method operates in two steps :

1. The procedure is statically transformed into a constraint system by the use of "Static Single Assignment" (SSA) form [23, 2, 9] and control-dependencies [12]. The result of this step is a set of constraints — called *Kset* — which is formed of :
   - the constraints generated for the whole procedure ;
   - the constraints that are specific to the selected point.
2. The constraint system *Kset* is solved to check whether at least one feasible path which goes through the selected point exists. Finally, test data corresponding to one of these paths are generated.

The key point of this method is to take advantages of current constraint techniques to solve the generated constraint system. In particular, global constraints are used in a preliminary step to detect some of the non-feasible parts of the control structures and partial consistency techniques are employed to reduce the domains of possible values of the test data. Search methods based on the combination of both enumeration and inference processes are used in the final step to identify test data. Furthermore, these techniques offer a flexible way to define and to solve new constraints on values of possible test data.

A prototype implementation of this method has been developped on a restricted subset of the C language.

Outline of the paper : the second section presents the generation of *Kset* while the third section is devoted to the resolution techniques. The fourth section describes the prototype implementation while the fifth section provides a detailed analysis of a non-trivial example that has been successfully treated with our method.

2 GENERATION OF THE CONSTRAINT SYSTEM

Application of our method is limited to a structured subset of a procedural language. Unstructured statements such as "*goto-statement*" are not handled in our framework because they introduce non-controled exits of loops and backward control flow.
Pointer aliasing, dynamic allocated structures, function's pointer involve difficult problems to solve in the frame of a static analysis. In this paper, we assume that programs avoid such constructions. The treatement of basic types such as char and floating point numbers is not presented. A few words in the fourth section are devoted to the extension of our method to these types.

The generation of the constraint system *Kset* is done in three steps :

1. Generation of the "Static Single Assignment" form ;
2. Generation of a set of constraints corresponding to the procedure p, called $pKset(p)$;
3. Generation of a set of constraints corresponding to the control-dependencies of a selected point n, called $cKset(n)$.

Kset is defined as :

$$Kset(p,n) \stackrel{def}{=} pKset(p) \cup cKset(n)$$

Now, let us introduce some basics used in the rest of the paper.

2.1 Basics

A procedure control flow graph (V, E, e, s) [1] is a connected oriented graph composed by a set of vertices V, a set of edges E and two particular nodes, e the unique

```
int f(int i)
      int j ;
1.    j := 1 ;
2.    while ( i ≠ 0 )
            do
3a.               j := j * i ;
3b.               i := i − 1 ;
            od ;
4.    if ( j = 2 )
5.          then i := 2 ;
      fi ;
6.    return j ;
```

Figure 1: Example 1

entry node, and s the unique exit node. Nodes represent the basic blocks which are sets of statements executed without branching and edges represent the possible branching between basic blocks. For instance, consider the procedure[1] given in figure 1, which is designed to compute the factorial function, and its control flow graph (CFG) shown in figure 2.

A *point* is either a node or an edge in the CFG. A *path* is a sequence $< v_i, \ldots, v_j >$ of consecutive nodes (edge connected) in (V, E, e, s). A *control flow path* is a path $< v_i, \ldots, v_j >$ in the CFG, where $v_i = e$ and $v_j = s$. A path is *feasible* if there exists at least one test datum on which the path is executed, otherwise it is *non-feasible*. For instance, the control flow path $< 1, 2, 4, 5, 6 >$ in the CFG of example 1 is non-feasible.

A node v_1 is *post-dominated* [12] by a node v_2 if every path from v_1 to s in (V, E, e, s) (not including v_1) contains v_2.
A node v_2 is *control-dependent* [12] on v_1 iff 1) there exists a path P from v_1 to v_2 in (V, E, e, s) with any v in $P \setminus \{v_1, v_2\}$ post-dominated by v_2 ; 2) v_1 is not post-dominated by v_2. For example, block 5 is control-dependent on block 4 in the CFG of example 1.

2.2 SSA Form

Most procedural languages allow destructive updating of variables ; this leads to the impossibility to treat a program variable as a logical variable. Initially proposed for the optimisation of compilers [2, 23], the "Static Single Assignment" form [9] is a semantically equivalent version of a procedure on which every variable has a unique definition and every use of a variable is reached by this definition. The SSA form of a linear sequence of code is obtained by a simple renaming ($i \longrightarrow i_0,\ i \longrightarrow i_1, \ldots$) of the variables. For the control structures, SSA form introduces special assign-

[1] For all the examples throughout the paper, a clear abstract syntax is used to indicate that our method is not designed to a particular language

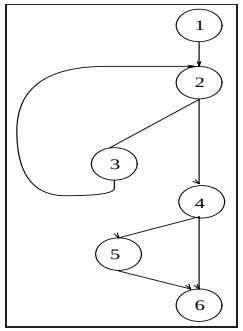


Figure 2: Control flow graph of example 1

```
int f(int i0)
      int j0 ;
1a.   j0 := 1 ;
      /* Heading */
1b.   j2 := φ(j0, j1) ;
1c.   i2 := φ(i0, i1) ;
2.    while (i2 ≠ 0 )
      do
3a.               j1 := j2 * i2 ;
3b.               i1 := i2 − 1 ;
      od
4.    if (j2 = 2)
5.                then i3 := 2 ;
      fi
6.    i4 := φ(i3, i2) ;
      return (j2) ;
```

Figure 3: SSA Form of example 1

ments, called ϕ-functions, in the junction nodes of the CFG. A ϕ-function returns one of its arguments depending on the control flow. Consider the *if-statement* of the SSA form of example 1 in figure 3 ; the ϕ-function of statement 6 returns i_3 if the flow goes through the *then-part* of the statement, i_2 otherwise. For some more complex structures, the ϕ-functions are introduced in a special heading of the loop (as in the *while-statement* in figure 3). SSA Form is built by using the algorithm given in [5], which is designed to treat structured programs in one parsing step.
For convenience, a list of ϕ-assignments will be written with a single statement :
$x_2 := \phi(x_1, x_0), \ldots, z_2 := \phi(z_1, z_0) \Longleftrightarrow \vec{v_2} := \phi(\vec{v_1}, \vec{v_0})$

2.3 Generation of *pKset*

pKset(p) is a set of both atomic and global constraints associated with a procedure p.

Informally speaking, an atomic constraint is a relation between logical variables. Global constraints are designed to handle more efficiently set of atomic constraints. For instance, global constraint ELEMENT/3 [2] : ELEMENT(k, L, v) constraints the kth argument of the list L to be equal to v.

Let us now present how *pKset* is generated. The method is driven by the syntax. Each subsection, which is devoted to a particular construction, presents the generation technique.

2.3.1 *Declaration*

The variables of a procedure are either input variables or local variables. Parameters and global variables are considered as input variables while the other variables are considered as local. Each variable x which has a basic type declaration, is translated in atomic constraint of the form : $x \in [Min, Max]$ where Min (resp. Max) is the minimum (resp. maximum) value depending on the current implementation. An array declaration is translated into a list of variables of the same type while a record is translated into a list of variables of different types.
A specific variable, named "OUT", is devoted to the output value of the procedure.

2.3.2 *Assignment and Decision*

Elementary statements, such as assignments and expressions in the decisions are transformed into atomic constraints. For instance, the assignment of statement 3a in example 1 generates the constraint $j_1 = j_2 * i_2$. The decision of statement 2 generates $i_2 \neq 0$. A basic block is translated into a conjunction of such constraints. For example, statements 3a and 3b generate $j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1$.

[2] where /3 denotes the arity of the constraint

2.3.3 *Conditional Statement*

The conditional statement *if_then_else* is translated into global constraint ITE/3 in the following way :

$$pKset(\textbf{if } \mathrm{d} \textbf{ then } \mathrm{s1} \textbf{ else } \mathrm{s2} \textbf{ fi } \vec{v_1} := \phi(\vec{v_2}, \vec{v_3})) = \mathrm{ITE}(pKset(d), pKset(s1) \wedge \vec{v_1} = \vec{v_2}, pKset(s2) \wedge \vec{v_1} = \vec{v_3})$$

This constraint denotes a relation between the decision and the constraints generated for the *then*- and the *else*-parts of the conditional. Note that ϕ-assignments are translated in simple equality constraints. The operational semantic of the constraint ITE/3 will be made explicit in section 3.2.

2.3.4 *Loop Statement*

The loop statement *while* is also translated in a global constraint W/5. Informally speaking, this constraint states that as long as its first argument is true, the constraints generated for the body (fifth argument) of the *while statement* are true for the required data.

$$pKset(\vec{v_2} := \phi(\vec{v_0}, \vec{v_1}) \ \ \textbf{while } \mathrm{d} \textbf{ do } s \textbf{ od}) = \mathrm{W}(pKset(d),\ \vec{v_0},\ \vec{v_1},\ \vec{v_2},\ pKset(s))$$

The generated constraint requires three vectors of variables $\vec{v_0}, \vec{v_1}, \vec{v_2}$. $\vec{v_0}$ is a vector of variables defined before the *while-statement*. $\vec{v_1}$ is the vector of variables defined inside the body of the loop and $\vec{v_2}$ is the vector of variables referenced inside and outside the *while-statement*. Note here that the ϕ-assignments are only used to identify the vectors of variables.

The operational semantics of the constraint W/5 will also be given in section 3.2.

2.3.5 *Array and Record*

Both arrays and records are treated as list of variables, therefore we only present the generation of *pKset* on arrays.

Reference of an array is provided in the SSA Form by a special expression [9] : *access*. The evaluation of *access(a,k)* statement is the kth element of a noted v.

For the definition of an array, the special expression *update* is used [9]. *update(a,j,w)* evaluates to an array a_1 which has the same size as a and which has the same elements as a, except for j where value is w.

Both expressions *access* and *update* are treated with the constraint ELEMENT/3 :

$$pKset(\ \text{v}:= access(a,k)) = \{\text{ELEMENT}(k,a,v)\}$$

$$pKset(a_1 := update(a,j,w))$$
$$= \bigcup_{i \neq j} \{\text{ELEMENT}(i,a,v) \wedge \text{ELEMENT}(i,a_1,v)\}$$
$$\cup \{\text{ELEMENT}(j,a_1,w)\}$$

2.4 Generation of *cKset*

cKset(n) is a set of constraints associated with a point n in the CFG. It represents the necessary conditions under which a selected point is executed. These conditions are precisely the control-dependencies on the selected point. *cKset(n)* is then the set of constraints of the statements and the branches on which n is control-dependent. For example, node 5 is control-dependant on node 4 then : $cKset(5) = \{j_2 = 2\}$

2.5 Example

For the procedure given in figure 1 and the statement 5, the following sets are obtained :
$pKset(f) =$
$\{\ j_0 = 1,$
$\text{W}(i_2 \neq 0, (i_0,j_0),(i_1,j_1),(i_2,j_2),$
$j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1),$
$\text{ITE}(j_2 = 2, i_3 = 2 \wedge i_4 = i_3, i_4 = i_2),$
$OUT = j_2\ \}$

$cKset(5) = \{j_2 = 2\}$

$Kset(f,5) = pKset(f) \cup cKset(5)$

3 SOLVING THE CONSTRAINT SYSTEM AND GENERATION OF TEST DATA

Constraint programming has emerged in the last decade as a new tool to address various classes of combinatorial search problems. Constraint systems are inference systems based on such operations as constraint propagation, consistency and entailment. Inference is based on algorithms which propagate the information given by one constraint to others constraints. These algorithms are usually called partial consistency algorithms because they remove part of inconsistent values from the domain of the variables. Altough these approximation algorithms sometimes decide inconsistency, it is usually necessary to combine the resolution process with a search method. Informally speaking, search methods are intelligent enumeration process.
For a survey on Constraint Solving and Constraint Logic Programming, see [14] and [17].

Let us first introduce some basics notations on constraint programming required in the rest of the paper. These notations are extracted from [15].

A constraint system is consistent if it has at least one solution, i.e. if there exists at least one variable assignment which respects all the constraints. More formally, a set of constraints σ is called a *store* and the store is consistent if :

$$\models (\exists)\sigma$$

where $(\exists)\phi$ denotes the existential closure of the formula ϕ.
Entailment test checks out the implication of a constraint by a store. For example,

$$x \geq 0 \text{ is entailed by } \{x = y^2\}$$

The entailment test of the constraint c by the store σ is noted :

$$\models (\forall)(\sigma \implies c)$$

where $(\forall)\phi$ denotes the universal closure of ϕ.
Both consistency and entailment tests are NP-complete problems in the general case. For this reason, implementations of these tests are based on two approximations : domain-consistency and interval-consistency.

3.1 Local consistency

Associated with each input variable x_i is both a domain $D_i \in \mathbb{Z}$ and an interval $D_i^* = [min(D_i), max(D_i)]$. A constraint $c(x_1,\ldots,x_n)$ is a n-ary relation between variables $(x_1,\ldots,x_n)$ which denotes a subset of $\mathbb{Z}^n$.

Domain-consistency also called arc-consistency removes values from the domains and Interval-consistency only reduces the lower and upper bounds on the domains. Both are applied in a subtle combination by the constraint solver. Intuitivelly, when the domains contain a small number of values, domain-consistency is applied. Interval-consistency is applied on large domains. Precise definitions of these local consistencies are now given :

```
int g(int x,int y )
      int z ;
      int t ;
1a.   z := x * y ;
1b.   t := 2 * x ;
2.    if (z ≤ 8)
            then
3a.               t := t − y ;
3b.               if (t = 1 ∧ x > 1 )
4.                      then ...
```

Figure 4: Example 2

Definition 1. *(domain-consistency) [15]*
A constraint c is domain-consistent if for each variable x_i and value $v_i \in D_i$ there exists values $v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_n$ in $D_1,\ldots,D_{i-1},D_{i+1},\ldots,D_n$ such that $c(v_1,\ldots,v_n)$ holds. A store σ is domain-consistent if for every constraint c in σ, c is domain-consistent.

Definition 2. *(interval-consistency) [15]*
A constraint c is interval-consistent if for each variable x_i and value $v_i \in \{min(D_i), max(D_i)\}$ there exist values $v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_n$ in $D_1^*,\ldots,D_{i-1}^*,D_{i+1}^*,\ldots,D_n^*$ such that $c(v_1,\ldots,v_n)$ holds.

A local treatment is associated to each constraint. The corresponding algorithm is able to check out both domain- and interval- consistencies for this constraint. The inference engine propagates the reductions provided by this algorithm on the other constraints. The propagation iterates until a fixpoint is reached. Informally speaking, a fixpoint is a state of the domains where no more prunnings can be performed.

Let us illustrate how interval-, domain- consistency and the inference engine may reduce the domains of possible values of test data on the example 2 given in figure 4. Consider the problem of automatic test data generation for statement 4.

Parameters are of non-negative integer type. The following set is provided :

$Kset(g,4) = \{x_0, y_0 \in [0, Max], z_0 = x_0 * y_0,\ t_0 = 2 * x_0,\ z_0 \leq 8,\ t_1 = t_0 - y_0,\ t_1 = 1,\ x_0 > 1\}$

and the following resolution process is performed :

$z_0 = x_0 * y_0$ leads to $z_0 \in [0, Max]$
$t_0 = 2 * x_0$ leads to $t_0 \in [0, Max]$
$z_0 \leq 8$ leads to $z_0 \in [0, 8]$
$t_1 = 1$ leads to $t_1 \in \{1\}$
$x_0 > 1$ leads to $x_0 \in [2, Max]$
$z_0 = x_0 * y_0$ leads to $x_0 \in [2, 8]$ and $y_0 \in [0, 4]$
$t_0 = 2 * x_0$ leads to $t_0 \in [4, 16]$
$t_1 = t_0 - y_0$ leads to $y_0 \in \{3, 4\}$ and $t_0 \in \{4, 5\}$
$t_0 = 2 * x_0$ leads to $x_0 \in \{2\}$ and $t_0 \in \{4\}$
$t_1 = t_0 - y_0$ leads to $y_0 \in \{3\}$

Finally, $(x_0 = 2, y_0 = 3)$ corresponds to the unique test datum on which statement 4 in the program of figure 4 can be executed.

3.2 Global Constraints Definitions

For atomic constraints and some global constraints, the local treatment is directly implemented in the constraint solver. However, for user-defined global constraint, it is necessary to provide the algorithm. The key point of our approach resides in the use of such global constraints to treat the control structures of the program. The global constraints are used to propagate information on inconsistency in a preliminary step of the resolution process.

3.2.1 *Entailment Test Implementation*

The entailment test is used to construct these global constraints. The implementation of entailment test may be done as a proof by refutation. A constraint is proved to be entailed by a store if there is no variable assignment respecting both the store and the negation of the constraint.

The operational semantic of the user-defined global constraints is designed with properties which are "guarded" by entailment tests. Such properties are expressed by constraints added to the store. We have introduced in the section 2.3 two global constraints : ITE/3 and W/5. Let us give now their definitions.

3.2.2 *ITE/3*

Definition 3. (ITE/3)
ITE$(c,\ \{c_1 \wedge \ldots \wedge c_p\},\ \{c'_1 \wedge \ldots \wedge c'_q\})$

- *if* $\models (\forall)(\sigma \implies c)$ *then* $\sigma := \sigma \cup \{c_1 \wedge \ldots \wedge c_p\}$
- *if* $\models (\forall)(\sigma \implies \neg c)$ *then* $\sigma := \sigma \cup \{c'_1 \wedge \ldots \wedge c'_q\}$
- *if* $\models (\forall)(\sigma \implies \neg(c_1 \wedge \ldots \wedge c_p))$ *then* $\sigma := \sigma \cup \{\neg c \wedge c'_1 \wedge \ldots \wedge c'_q\}$
- *if* $\models (\forall)(\sigma \implies \neg(c'_1 \wedge \ldots \wedge c'_q))$ *then* $\sigma := \sigma \cup \{c \wedge c_1 \wedge \ldots \wedge c_p\}$

The first two features of this definition express the operational semantic of the control structure *if_then_else*. The last ones are added to identify non-feasible parts formed by one of the two branches of the control

structure. Consider for example :

$$\text{ITE}(i_0 \neq 0, i_1 = i_0 - 1 \wedge i_2 = i_1, i_2 = 1)$$

Suppose that the store contains $i_2 = 0$; when applying the fourth feature of the ITE constraint we have to consider the consistency of the following set : $\{i_2 = 0\} \cup \{i_2 = 1\}$ It is inconsistant, meaning that the *else*-part of the statement is non feasible. Then, the constraints $i_0 \neq 0 \wedge i_1 = i_0 - 1 \wedge i_2 = i_1$ are added to the store.

3.2.3 *W/5*

The *while-statement* combines looping and destructive assignments. Hence W/5 behaves as a constraint generation program.

When evaluating W/5, it is necessary to allow the generation of new constraints and new variables. A substitution $subs(\vec{v_1} \leftarrow \vec{v_2}, c)$ is a mechanism which generates a new constraint having the same structure as c but where variables vector $\vec{v_1}$ has been replaced by vector $\vec{v_2}$. The following example illustrates this mechanism : if $\vec{v_1} = (x_1, y_1)$ and $\vec{v_2} = (x_2, y_2)$ then
$subs(\vec{v_1} \leftarrow \vec{v_2}, x_1 + y_1 = 3)$ is $(x_2 + y_2 = 3)$

W/5 is now formally defined :

Definition 4. (W/5)
$\text{W}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, c_1 \wedge \ldots \wedge c_p)$

- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, c))$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, c_1 \wedge \ldots \wedge c_p) \wedge$
  $\text{W}(c, \vec{v_1}, \vec{v_3}, \vec{v_2},$
  $subs(\vec{v_1} \leftarrow \vec{v_3}, c_1 \wedge \ldots \wedge c_p))\}$
- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, \neg c))$ *then*
  $\sigma := \sigma \cup \{\vec{v_2} = \vec{v_0}\}$
- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, \neg(c_1 \wedge \ldots \wedge c_p)))$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, \neg c) \wedge \vec{v_2} = \vec{v_0}\}$
- *if* $\models (\forall)(\sigma \implies \vec{v_2} \neq \vec{v_0})$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, c) \wedge$
  $subs(\vec{v_2} \leftarrow \vec{v_0}, c_1 \wedge \ldots \wedge c_p) \wedge$
  $\text{W}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, subs(\vec{v_1} \leftarrow \vec{v_3}, c_1 \wedge \ldots \wedge c_p))\}$

The first two features represent the operational semantic of the *while*-statement. As for the ITE/3 constraint, the other features identify non-feasible part of the structure. The third one is applied if it can be proved that the constraints of the body of the loop are inconsistent with the current store. This means the body cannot be executed even once, the output vector of variables $\vec{v_2}$ is then equated with the input vector $\vec{v_0}$. In the opposite, if $\vec{v_2} = \vec{v_0}$ is inconsistent in the current store, the fourth feature is applied meaning that the body of the loop is executed at least once.

Let us illustrate the treatment of W/5 on the *while*-statement of example 1 :

Suppose that the store contains $\{j_0 = 1, j_2 = 2\}$; when testing the consistency of

$$\text{W}(i_2 \neq 0, (i_0, j_0), (i_1, j_1), (i_2, j_2), \\ j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1)$$

the fourth feature is applied twice and then gives the following store :

$$\{j_0 = 1, j_2 = 2, j_2 = j_1 * i_1, i_2 = i_1 - 1, \\ j_1 = j_0 * i_0, i_1 = i_0 - 1, i_2 = 0\}$$

Finally, $(i_0 = 2)$ is obtained.

3.3 Complete Resolution of the Example

Consider again the example of figure 1 and the problem of generating a test data on which a feasible path going through statement 5 is executed. The *Kset* provided by the first step of our method is :

$$Kset(f, 5) = pKset(f) \cup cKset(5) = \\ \{ \ j_0 = 1, \\ \text{W}(i_2 \neq 0, (i_0, j_0), (i_1, j_1), (i_2, j_2), \\ j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1), \\ \text{ITE}(j_2 = 2, i_3 = j_2 \wedge i_4 = i_3, i_4 = i_2), \\ OUT = j_2\} \cup \{j_2 = 2\}$$

The loop is executed twice, generating the following store :

$$\{j_0 = 1, i_0 \neq 0, i_1 \neq 0, i_2 = 0, i_2 = i_1 - 1, i_1 = i_0 - 1, j_2 = j_1 * i_1, j_1 = j_0 * i_0, j_2 = 2, i_3 = j_2, i_4 = i_3, OUT = j_2\}$$

Interval consistency is applied to solve the system, and yields to $i_0 = 2$. This is the unique test data on which statement 5 may be executed.

3.4 Search Process

Of course, local consistencies are incomplete constraint solving techniques [22]. The store of constraints can be domain-consistent though there is no solution in the domains (i.e. the store is inconsistent). Let us give an example of a classical pitfall of these techniques :
$x, y, z \in \{0, 1\}, \quad \sigma = \{x \neq y, y \neq z\}$
Testing $\models (\forall)(\sigma \implies (x = z))$ fails because the store $\{x \neq y, y \neq z, x \neq z\}$ is domain-consistent.

In order to obtain a solution, it is necessary to enumerate the possible values in the restricted domains [22, 15]. This process is incremental. When a value v is chosen in the domain D_x of the variable x, the constraint $(x = v)$ is added to the store and propagated. This may reduce the domains of the other variables. This process is repeated until either the domain of all variables is reduced to a single value or the domain of some variable becomes empty. In the former case, we obtain a solution of the test data generation problem, whereas in the latter we must backtrack and try another value $(x = w)$ until $D_x = \emptyset$.

In general, there are many test data on which a selected point is executed. As claimed in the Introduction, constraint solving techniques provide a flexible way to choose test data. The search process can be user-directed by adding new constraints on the input variables of the procedure. Our framework provides an elegant way to handle such constraints. These constraints are propagated by the inference engine as soon as they induce a reduction on the domains. Furthermore, these additional constraints may be used to insure that the generated input data are "realistic". They may have one of the two following forms :

- constraints on domains (for example $x_0 \in [-3, 17]$) ;
- constraints between variables (for example $y_0 > x_0$ meaning that a parameter y_0 of a procedure is strictly greater than another one x_0).

It is also possible to guide the search process with some well-known heuristics. For example :

- to select the variable with the smallest domain (first-fail principle) ;
- to select the most constrained variable ;
- to bissect the domains ($x \in [a, b]$ is transformed into $(D_x = [a, a + b/2]$ or $D_x = [a + b/2, b])$).

4 IMPLEMENTATION

INKA, a prototype implementation has been developed on a structured subset of language C. The extension to control structures such as *do-while* and *switch* statement is straightforward. Characters are handled in the same way as integer variables. Floating point numbers do not introduce new difficulties in the constraints generation process, but they require another solver. Although the domains remain finite, it is of course not possible to enumerate all the values of a floating point variable. Resolution of the constraint system is therefore more problematic. References on these solvers can be found in [17]. The extension of our method to pointer variables falls into two classical problems of static analysis : the aliasing problem and the analysis of dynamic allocated structures.

INKA includes 5 modules :

- A C Parser
- A generator of SSA form and control-dependencies
- A generator of *Kset*
- A constraints solver
- A search process module

The constraint solver is provided by the CLP(FD) library of Sicstus Prolog 3.5 [6].

5 EXAMPLE

We present now the results of our method on a non-trivial example adapted from [11] : the SAMPLE program given in figure 5. For the sake of simplicity, it is written in the abstract syntax used in this paper. Size of array have been reduced to 3 for improving the presentation.

Consider the problem of automatic test data generation to reach node 13.

INKA has generated the *Kset(SAMPLE,13)* constraint system. The following set of constraints on domains are added :

$$a[1], a[2], a[3], b[1], b[2], b[3], target \in [1, 9]$$

Table 1 reports only the results of the constraint solver and search process module. Experiments are made on a Sun Sparc 5 workstation under Solaris 2.5

First experiments concern the search of solutions without adding any kind of constraints on input data. The line 1 of the table 1 indicates the time required to obtain the first solution and all solutions of the problem. The exact test data is provided in the former case while the number of solutions is only provided in the later one.

Then, we have considered that the user wants the input data to satisfy the additional constraint :

$$a[3]^2 = a[1]^2 + a[2]^2$$

```
int sample(int a[3], int b[3], int target)
    int i, fa, fb, out ;
1a. i := 1 ;
1b. fa := 0 ;
1c. fb := 0 ;
2.  while (i ≤ 3)
        do
3.          if (a[i] = target)
4.              then fa := 1 ;
            fi ;
5.          i := i + 1 ;
        od
6.  if (fa = 1)
        then
7a.         i := 1 ;
7b          fb := 1 ;
8.          while (i ≤ 3)
                do
9.                  if (b[i] ≠ target)
10.                     then fb := 0 ;
                    fi ;
11.                 i := i + 1 ;
                do ;
    fi ;
12. if (fb = 1)
13.     then out := 1 ;
14.     else out := 0 ;
    fi ;
15. return out ;
```

Figure 5: Program SAMPLE

Second line reports the results of generation when the additional constraint is checked out after the search process and the third line reports the results when the constraint is added to the current store and propagated.

A first fail enumeration heuristic has been used for these experiments. *Test data* are given in vector form $(a[1], a[2], a[3], b[1], b[2], b[3], target)$ and *CPU time* is the time elapsed in the constraint solving phase. Note that a complete enumeration stage would involve to try $9^7 = 4782969$ values.

These experiments are intended to show what we have called the flexible use of constraints. First, the CPU time elapsed in the first and second experiments are approximatively the same to obtain all the solutions. In both cases, the search process has enumerated all the possible values in the reduced domains. The only difference is that, in the second case, the added constraint has been checked out after the enumeration step. This illustrate a generate and test approach. On the contrary, note that the results presented in the third line of table 1 show an important improvement factor due to the use of the additional constraint in the resolution process. In the third case, the additional constraint is used to prune the domains and thus the time elapsed in the search process module is dramatically reduced.

Of course, further experiments are needed to show the effectiveness of our approach and to compare the method with other approaches.

Table 1: Results

First solution	CPU time	All solutions	CPU time
(1,1,1,1,1,1,1)	1.0s	1953 solutions	287s
(3,4,5,3,3,3,3)	53s	(3,4,5,3,3,3,3), (3,4,5,4,4,4,4), (3,4,5,5,5,5,5), (4,3,5,3,3,3,3), (4,3,5,4,4,4,4), (4,3,5,5,5,5,5)	292s
(3,4,5,3,3,3,3)	1.3s	(3,4,5,3,3,3,3), (3,4,5,4,4,4,4), (3,4,5,5,5,5,5), (4,3,5,3,3,3,3), (4,3,5,4,4,4,4), (4,3,5,5,5,5,5)	2.4s

6 CONCLUSION

In this paper, we have presented a new method for the automatic test data generation problem. The key point of this approach is the early detection of some of the non-feasible paths by the global constraints and thus the reduction of the number of trials required for the generation of test data. First experiments on a non-trivial example made with a prototype implementation tend to show the flexibility of our method. Future work will be devoted to the extension of this method to pointer variables and experimentations with floating point numbers ; an experimental validation on real applications is also forseen.

ACKNOWLEDGEMENTS

Patrick Taillibert and Serge Varennes gave us invaluable help on preliminary ideas to design the global constraints introduced. Thanks to Xavier Moulin for its helpful comments on earlier drafts of this paper.
This work is partially supported by A.N.R.T.
This research is part of the software testing project DEVISOR of Dassault Electronique.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, techniques and tools*. Addison-Wesley Publishing Company, Inc, 1986.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proc. of Symposium on Principles of Programming Languages*, pages 1–11, New York, January 1988. ACM.

[3] A. Bertolino and M. Marré. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, December 1994.

[4] R. Boyer, B. Elspas, and K. Levitt. SELECT - A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, June 1975.

[5] M. M. Brandis and H. Mőssenbőck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[6] M. Carlsson. *SICStus Prolog User's Manual, Programming over Finite Domains*. Swedish Institute in Computer Science, 1997.

[7] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[8] A. Coen-Porisini and F. de Paoli. Array Representation in Symbolic Execution. *Computer Languages*, 18(3):197–216, 1993.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] R. A. DeMillo and A. J. Offut. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, SE-17(9):900–910, September 1991.

[11] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation". *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9-3:319–349, July 1987.

[13] D. Hamlet, B. Gifford, and B. Nikolik. Exploring Dataflow Testing of Arrays. In *Proc. of the International Conference on Software Engineering*, pages 118–129, Baltimore, May 1993. IEEE.

[14] P. V. Hentenryck and V. Saraswat. Constraints Programming : Strategic Directions. *Constraints*, 2(1):7–34, 1997.

[15] P. V. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). In *LNCS 910*, pages 293–316. Springer Verlag, 1995.

[16] W. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, July 1977.

[17] J. Jaffar and M. J. Maher. Constraint Logic Programming : A Survey. *Journal of Logic Programming*, 20(19):503–581, 1994.

[18] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[19] B. Korel. A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance*, pages 311–317, San Diego, CA, November 1990. IEEE.

[20] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, august 1990.

[21] B. Korel. Automated Test Data Generation for Programs with Procedures. In *Proc. of ISSTA '96*, volume 21(3), pages 209–215, San Diego, CA, May 1996. ACM, SIGPLAN Notices on Software Engineering.

[22] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[23] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proc. of Symposium on Principles of Programming Languages*, pages 12–27, New York, January 1988. ACM.

[24] D. F. Yates and N. Malevris. Reducing The Effects Of Infeasible Paths In Branch Testing. In *Proc. of Symposium on Software Testing, Analysis, and Verification (TAV3)*, volume 14(8) of *Software Engineering Notes*, pages 48–54, Key West, Florida, December 1989.

Portée de l'article

L'article précédent propose la construction d'un modèle à contraintes des programmes C qui permet la résolution de certaines requêtes de génération de donnée de test avec les techniques standards de la Programmation par Contraintes sur les domaines finis [Hentenryck 93, Hentenryck 98]. En particulier, la propagation de contraintes, le filtrage par consistance locale (consistance de domaine et bornes) et quelques stratégies de recherche telles que "first-fail" ou "iterative domaine-splitting" sont utilisées. Dès sa publication et malgré sa naïveté, cet article a été utilisé comme source de comparaison par de nombreux travaux qui ont démarré à cette époque (cité plus de 200 fois – *Source: Publish or Perish, version 3.2, Juillet 2011*).

Dès la fin des années quatre-vingt-dix, Neelam Gupta de l'Université d'Arizona s'est montrée intéressée par l'utilisation des contraintes, dans la mesure où elle menait des travaux connexes sur l'utilisation de relaxation itérative pour la génération automatique de cas de test structurels [Gupta 00, Visvanathan 02]. De même, Willem Visser, à l'origine du moteur de vérification Java PathFinder [Visser 04] développé au centre Nasa Ames, s'est intéressé à nos travaux pour analyse et comparaison [C. Pasareanu 03], ainsi que Christian Cadar, alors qu'il développait l'outil EXE à l'Université de Stanford. En Europe, Nguyen Tran Sy et Yves Deville de l'Université Catholique de Louvain [Sy 01, Sy 03] ont cherché à adapter des consistances partielles pour la génération automatique de cas de test structurel en présence de calculs mixtes entiers-flottants. C'est également dans l'article commenté de ce chapitre qu'a été introduit pour la première fois l'utilisation de la forme statique à assignation unique (SSA) en génération automatique de tests. Cet article a ainsi été remarqué dans la communauté analyse statique, en particulier grâce à des gens comme Gregor Snelting [Snelting 06], ou Thomas Ball [Ball 05]. L'utilisation de la forme SSA pour la génération automatique de tests était réellement novatrice à cette époque et cette idée a fait florès depuis [Sy 01, Collavizza 08, Chung 09, Wotawa 10].

L'accueil de la communauté pour cet article nous a confortés, à l'époque, dans notre volonté de développer un outil de génération automatique de données de test pour un langage réaliste tel que C. Ce fut donc le début de la réalisation de l'outil InKa, mise au point chez Dassault Electronique, avec des idées développées en collaboration avec l'Université de Nice Sophia-Antipolis.

Chapter 2

Test logiciel à base de contraintes

Contexte

La réalisation de l'outil InKa a été particulièrement importante pour démontrer l'utilité des contraintes en génération automatique de données de test. En effet, les logiciels de test unitaire et d'intégration utilisés alors permettaient essentiellement d'automatiser l'exécution du test, mais en aucun cas de déterminer automatiquement les entrées permettant de sensibiliser certaines portions critiques du code source. Par ailleurs, la génération de cas de test structurels et plus généralement la vérification de logiciel n'était pas alors considérée alors une application standard de la *Programmation par Contraintes*.

L'article de ce chapitre fut le premier à présenter le prototype InKa, ainsi que les premières expériences relatives au modèle à contraintes sous-jacent. Nous y détaillons les consistances partielles utilisées, ainsi que la sémantique opérationnelle des combinateurs de contraintes utilisés pour représenter le flot de contrôle (i.e., `ite` et `w`). Un des points les plus intéressants concerne peut-être la description du traitement des contrainte–gardées avec un test d'implication par l'absurde. C'est aussi dans cet article que le traitement des tableaux avec la contrainte globale `element` est proposé, une idée simple qui est passée complètement inaperçue à l'époque.

Dans les années 2000, avec le logiciel InKa, nous caressions le projet de traiter exhaustivement un langage réaliste, utilisé dans le domaine de l'avionique embarqué (i.e., le langage C). Mais, il a bien fallut réviser ces prétentions devant la difficulté de la tâche. En effet, dès cette époque, nous avions identifié la problématique de la synonimie due aux pointeurs, références et tableaux, comme étant une difficulté majeure de l'analyse de code C. Le traitement efficace des appels de fonctions ou méthodes, y compris récursifs, était à inventer, ainsi que celui du polymorphisme par invocation de méthodes virtuelles. La problématique de l'analyse des calculs sur les nombres flottants était par contre bien connue dans d'autres communautés, mais la résolution de contraintes pour les flottants était inconnue.

Ces éléments se révélèrent être des difficultés majeures pour toutes les approches de génération automatique de données de test à base d'analyse de code source. D'autres éléments apparurent également comme le traitement des calculs modulaires silencieux en C, la modélisation d'une sémantique avec erreurs et exceptions, le traitement des structures dynamiques et des pointeurs en entrée des fonctions, les pointeurs de fonctions et le traitement de l'ordre supérieur, la modélisation des calculs bit-à-bits, ou encore le traitement des programmes au flot déstructuré.

A. Gotlieb, B. Botella, and M. Rueher. *A CLP framework for computing structural test data*. **In Proceedings of Computational Logic (CL'2000), LNAI 1891, pages 399–413, London, UK, Jul. 2000**

A CLP Framework for Computing Structural Test Data

Arnaud Gotlieb[1], Bernard Botella[1], and Michel Rueher[2]

[1] Thomson-CSF Detexis, Centre Charles Nungesser 2, av. Gay-Lussac
78851 Elancourt Cedex, France
{Arnaud.Gotlieb,Bernard.Botella}@detexis.thomson-csf.com
[2] Université de Nice–Sophia-Antipolis, I3S, ESSI, 930, route des Colles - B.P. 145
06903 Sophia-Antipolis, France
rueher@essi.fr — http://www.essi.fr/~rueher

Abstract. Structural testing techniques are widely used in the unit testing process of softwares. A major challenge of this process consists in generating automatically test data, i.e., in finding input values for which a selected point in a procedure is executed. We introduce here an original framework where the later problem is transformed into a CLP(FD) problem. Specific operators have been introduced to tackle this kind of application. The resolution of the constraint system is based upon entailment techniques. A prototype system — named INKA— which allows to handle a non-trivial subset of programs written in C has been developed. First experimental results show that INKA is competitive with traditional ad–hoc methods. Moreover, INKA has been used successfully to generate test data for programs extracted from a real application.

1 Introduction

Structural testing techniques are widely used in the unit or module testing process. Structural testing requires :

1. Identifying a set of statements in the procedure under test, the covering of which implies the coverage of some criteria (e.g., statement or branch coverage) ;
2. Computing test data so that each statement of the set is reached.

The second point —called ATDG[1] problem in the following— is the corner stone of structural testing since it arises for a wide range of structural criteria. The ATDG problem is undecidable in the general case since it can be reduced to the halting problem. Classical ad-hoc methods fall into three categories :

- Random test data generation techniques which blindly try values [Nta98] until the selected point is reached ;
- Symbolic-execution techniques [Kin76,DO93] which replace input parameters by symbolic values and which statically evaluate the statements along the paths reaching the selected point ;
- Dynamic methods [Kor90,FK96] which are based on actual execution of procedure and which use heuristics to select values, e.g. numerical direct search methods.

[1] Automatic Test Data Generation

The limit of these techniques mainly comes from the fact that they "follow one path" in the program and thus fail to reach numerous points in a procedure. A statement in a program may be associated with the set of paths reaching it, whereas a test datum on which the statement is executed follows a single path. However, there are numerous non-feasible paths, i.e., there is no input data for which such paths can be executed. Furthermore, if the procedure under test contains loops, it may contain an infinite number of paths.

We introduce here an original framework where the ATDG problem is transformed into a CLP problem over finite domains. Roughly speaking, this framework can be defined by the following three steps :

1. Transformation of the initial program into a CLP(FD) program with some specific operators which have been introduced to tackle this kind of application ;
2. Transformation of the selected point into a goal to solve in the CLP(FD) system ;
3. Solving the resulting constraint system to check whether at least one feasible control flow path going through the selected point exists, and to generate automatically test data that correspond to one of these paths.

The two first steps are based on the use of the "Static Single Assignment" form [CFR+91] and control-dependencies [FOW87]. They have been carefully detailed in [GBR98].

In this paper, we mainly analyze the third step : the constraint solving process. The key-point of our approach is the use of constraint entailment techniques to drive this process efficiently. In the proposed CLP framework test data can be generated without following one path in the program.

To validate this framework, a prototype system — named INKA— has been developed over the CLP(FD) library of Sicstus Prolog. It allows to handle a non-trivial subset of programs written in C. The first experimental results show that INKA overcomes random generation techniques and is competitive with other methods. Moreover, INKA has been used successfully to generate test data for programs extracted from a real application.

Before going into the details, let us illustrate the advantage of our approach on a very simple example.

1.1 Motivating Example

Let us consider the small toy–program given in Fig. 1. The goal is to generate a test datum, i.e. a pair of values for (x, y), for which statement 10 is executed. "Static Single Assignment" techniques and control-dependencies analysis yield

the following constraint system[2] :

$\sigma_1 = (x, y, z, t_1, t_2, u \in (0..2^{32}-1) \wedge$
$(z = x * y) \wedge (t_1 = 2 * x) \wedge (z \leq 8) \wedge (u \leq x) \wedge (t_2 = t_1 - y) \wedge (t_2 \leq 20)$

Variables t_1 and t_2 denote the different renaming of variable t.

```
int foo(int x, int y)
    int z, t, u ;
1.  { z = x * y ;
2.    t = 2 * x ;
3.    if (x < 4)
4.      u = 10 ;
      else
5.      u = 2 ;
6.    if (z ≤ 8)
7.      { if (u ≤ x)
8.          { t = t - y ;
9.            if (t ≤ 20)
10.             { ...
```

Fig. 1. Program foo

Local consistency techniques like *interval–consistency* [HSD98] cannot achieve any significant pruning : the domain of x will be reduced to $0..2^{16}-1$ while no reduction can be achieved on the domain of y. So, the search space for (x, y) contains $(2^{16}-1) \times (2^{32}-1)$ possible test data. However, more information could be deduced from the program. For instance, the following relations could be derived from the first `if_then_else` statement (lines 3,4,5) :

$(x \geq 4 \wedge u = 2)$ holds if $\neg(x < 4 \wedge u = 10)$ holds
$(x < 4 \wedge u = 10)$ holds if $\neg(x \geq 4 \wedge u = 2)$ holds

Entailment mechanisms allow to capture such information. Indeed, since $\neg(x < 4 \wedge u = 10)$ is entailed by $u \leq x$, we can add to the store the constraint $(x \geq 4 \wedge u = 2)$. Filtering $x \geq 4 \wedge u = 2 \wedge \sigma_1$ by *interval-consistency* reduces the domain of x to $4..11$ and the domain of y to $0..2$.
This example shows that entailment tests may help to drastically reduce the search space. Of course, the process becomes more tricky when several conditional statements and loop statements are inter-wound.

Outline of the Paper The next section introduces the notation and some basic definitions. Section 3 details how the constraint system over CLP(FD) is generated. Section 4 details the constraint solving process. Section 5 reports the first experimental results obtained with INKA, while section 6 discusses the extensions of our framework.

[2] In this context, an **int** variable has an unsigned long integer value, i.e. a value between 0 and $2^{32}-1$.

2 Notations and Basic Definitions

A *domain* in FD is a non-empty finite set of integers. A variable which is associated to a domain in FD is called a FD_variable and will be denoted by an upper-case letter. *Primitive constraints* in CLP(FD) are built with variables, domains, the $\in$ operator, arithmetical operators in $\{+, -, \times$ `div`, `mod` $\}$ [3] and the relations $\{>, \geq, =, \neq, \leq, <\}$. Note that the negation of a primitive constraint is also a primitive constraint. In the following, c possibly subscripted denotes exclusively a primitive constraint. A *constraint–store* σ is a conjunction of primitive and non-primitive constraints.

Non-primitive constraints are composed of combinators and guarded–constraints. *Combinators* are boolean combination of constraints. For example, the constraint `element`(I, L, V) which express that V is the Ith element in the list L is a combinator.
Guarded–constraints are built by using the blocking ask operator [HSD98] and are denoted $C_1 \longrightarrow C_2$, where C_1 and C_2 stand for constraints. C_1 is called the guard. The operational semantic of $C_1 \longrightarrow C_2$ is given by the following rules:

- the constraint $C_1 \longrightarrow C_2$ is removed and C_2 is added to σ when C_1 is entailed by σ;
- the constraint $C_1 \longrightarrow C_2$ is just removed when $\neg C_1$ is entailed by σ;
- the constraint $C_1 \longrightarrow C_2$ is suspended when neither C_1 nor $\neg C_1$ are entailed by σ;

Note that C_1 and C_2 are not restricted to be primitive and that checking whether $\neg C_1$ is entailed may require to compute the negation of a non-primitive constraint.

Entailment operations are based on partial consistencies. Two partial entailment tests have been introduced in [HSD98]: *domain-entailment* and *interval-entailment*. They are based upon *domain-consistency* and *interval-consistency*. Let $X_1, \ldots, X_n$ be FD_variables, let $D_1, \ldots, D_n$ be domains and let C be a constraint[4].

Definition 1 *(domain-consistency)*
A constraint C is domain-consistent if for each variable X_i and value $v_i \in D_i$ there exists values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $C(v_1, \ldots, v_n)$ holds. A store σ is domain-consistent if for every constraint C in σ, C is domain-consistent.

Interval consistency is based on an approximation of finite domains by finite sets of successive integers. More precisely, if D is a domain, D^* is defined by the set $\{\min(D), \ldots, \max(D)\}$ where $\min(D)$ and $\max(D)$ denote respectively the minimum and maximum values in D.

Definition 2 *(interval-consistency)*
A constraint C is interval-consistent if for each variable X_i and value $v_i \in$

[3] `div` and `mod` represent the Euclidean division and remainder
[4] we assume that all the constraints are implicitly defined on $X_1, \ldots, X_n$

$\{min(D_i), max(D_i)\}$ *there exist values* $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ *in* $D_1^*, \ldots, D_{i-1}^*, D_{i+1}^*, \ldots, D_n^*$ *such that* $C(v_1, \ldots, v_n)$ *holds. A store* σ *is interval-consistent if for every constraint* C *in* σ, C *is interval-consistent.*

The following relaxations of entailment are introduced in [HSD98]:

Definition 3 *(domain-entailment)*
A constraint $C(X_1, \ldots, X_n)$ *is domain-entailed by* $D_1, \ldots, D_n$ *iff, for all values* $v_1, \ldots, v_n$ *in* $D_1, \ldots, D_n$, $C(v_1, \ldots, v_n)$ *holds.*

Definition 4 *(interval-entailment)*
A constraint $C(X_1, \ldots, X_n)$ *is interval-entailed by* $D_1, \ldots, D_n$ *iff, for all values* $v_1, \ldots, v_n$ *in* $D_1^*, \ldots, D_n^*$, $C(v_1, \ldots, v_n)$ *holds.*

We introduce here another partial entailment test which is based on refutation :

Definition 5 *(abs-entailment)*
A constraint C *is abs-entailed by a store* σ *iff, filtering* $\sigma \wedge \neg C$ *by domain-consistency or interval-consistency yields an empty domain.*

3 Generation of the Constraint System

Let P be a single procedure written in an imperative language, let n be a point (either a statement or a branch) in P. Solving the ATDG problem requires to compute a vector of input[5] values of P such that n is executed.

For the sake of simplicity, we first introduce the constraint system generation technique for an `array_if_while` language over integers. Procedure calls are handled in our framework but we assume that there is only one mechanism for passing arguments : the call–by–value mechanism. Programs must be well-structured and must avoid floating-point variables. A procedure is assumed to have a single return statement.

Next subsection recalls the general principles of the "Static Single Assignment" form [CFR+91]. The following subsections detail the transformation process of a program under SSA form into a CLP program.

3.1 Static Single Assignment Form

The SSA form is a version of a procedure on which every variable has a unique definition and every use of a variable is reached by this definition. The SSA form of a basic block is obtained by a simple renaming ($i = i + 1$ yields $i_2 = i_1 + 1$). For the control structures, SSA form introduces special assignments, called ϕ-functions, to merge several definitions of the same variable. For example, the SSA form of the `if_then_else` statement is illustrated in the top of Fig. 2. The ϕ-function of the statement $u_3 = \phi(u_1, u_2)$ returns one of its argument : if the flow comes from the *then*- part then the ϕ-function returns u_1, otherwise it

[5] An input variable is either a formal parameter or a referenced global variable

if $(x < 4)$ $u = 10$; **else** $u = 2$;	**if** $(x < 4)$ $u_1 = 10$; **else** $u_2 = 2$; $u_3 = \phi(u_1, u_2)$;
$j = 1$; **while** $(j * u \leq 16)$ $j = j + 1$	$j_1 = 1$; /* Heading - while */ $j_3 = \phi(j_1, j_2)$; **while** $(j_3 * u_3 \leq 16)$ $j_2 = j_3 + 1$;

Fig. 2. SSA form of control statements

returns u_2.
For other structures such as loops, the ϕ-functions are introduced in a special heading which is executed at every iteration. The ϕ-functions work as usual : this explains the counter-intuitive renaming of variables (see Fig. 2).

For convenience, a list of ϕ-functions will be written with a single statement : $x_2 := \phi(x_1, x_0), \ldots, z_2 := \phi(z_1, z_0) \Longleftrightarrow \boldsymbol{v_2} := \phi(\boldsymbol{v_1}, \boldsymbol{v_0})$ where $\boldsymbol{v_i}$ stands for a vector of variables.

3.2 Generation of the CLP Program

The basic idea is to translate each statement of the SSA form into a primitive constraint or a combinator, in order to build a CLP program. A clause is generated for each procedure P of the program. The head of the clause has several arguments :

- a list of FD_variables associated with the parameters of P ;
- a list of FD_variables associated with the referenced globals of P ;
- a list of FD_variables associated with the local variables used inside the decisions of P ;
- a list of FD_variables associated with the globals defined inside P ;
- a single FD_variable associated with the expression returned by P.

Now, let us detail the transformation process.

Declaration. A type declaration of a variable x_i is translated into a primitive constraint of the form : $X_i \in Min_T..Max_T$ where Min_T (resp. Max_T) is the minimum (resp. maximum) value of the type T. Such a constraint prevents overflows of values, a condition which is required to generate a test datum on which a selected point is reached.

Array. SSA form provides special expressions to handle arrays : $access(a_0, k)$ which evaluates to the kth element of a_0, and $update(a_0, j, w)$ which evaluates to an array a_1 which has the same size and the same elements as a_0, except for j where value is w. *access* and *update* expressions are transformed into `element/3` constraints:

- $v = access(a_0, k)$ is translated into $\texttt{element}(K, A_0, V)$;
- a definition statement $a_1 = update(a_0, j, w)$ is translated into $\texttt{element}(J, A_1, W) \quad \bigwedge_{I \neq J}(\texttt{element}(I, A_0, V) \wedge \texttt{element}(I, A_1, V))$.

Conditional. The *if_then_else* statement is treated by using a combinator, called `ite/3`. For example, the *if_then_else* statement of Fig. 2 is translated into : $\texttt{ite}(X < 4, U_1 = 10 \wedge U_3 = U_1, U_2 = 2 \wedge U_3 = U_2)$. The conditional statement express an exclusive disjunction between two paths. So, $\texttt{ite}(c, C_1 \wedge \ldots \wedge C_n, C'_1 \wedge \ldots \wedge C'_m)$ holds iff $(c \wedge C_1 \wedge \ldots \wedge C_n)$ or $(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$ holds, where c is a primitive constraint, and $C_1, \ldots, C_n, C'_1, \ldots, C'_m$ are primitive or non-primitive constraints. The operational semantic of combinator `ite/3` is based on the following rules:

Definition 6 `ite/3` *(operational semantic)*
$\texttt{ite}(c, C_1 \wedge \ldots \wedge C_n, C'_1 \wedge \ldots \wedge C'_m)$ *is reduced to the four following guarded–constraints :*

- $c \longrightarrow C_1 \wedge \ldots \wedge C_n$
- $\neg c \longrightarrow C'_1 \wedge \ldots \wedge C'_m$
- $\neg(c \wedge C_1 \wedge \ldots \wedge C_n) \longrightarrow (\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$
- $\neg(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m) \longrightarrow (c \wedge C_1 \wedge \ldots \wedge C_n)$

The first two guarded–constraints result from the operational semantic of the *if_then_else* statement in an imperative language. The last two are introduced to allow a more effective pruning. c and $\neg c$ are included in the guards to facilitate the detection of inconsistencies by *abs-entailment* (see section 4).

Loop. Unlike the conditional, the *while* statement under SSA form cannot be translated directly. A *while* statement in SSA form is of the general form : $\boldsymbol{v_2} = \phi(\boldsymbol{v_0}, \boldsymbol{v_1})$ *while* (c) $\{C_1; \ldots; C_p\}$ where $\boldsymbol{v_0}$ is the vector of input variables of the *while*, $\boldsymbol{v_1}$ is the vector of variables defined inside the body of the *while*, and $\boldsymbol{v_2}$ is the vector of variables used inside and outside the *while*. This statement is transformed into a $\texttt{w}(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \cdots \wedge C_p)$ combinator, which is a constraint generation program.
$\texttt{w}(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \ldots \wedge C_p)$ holds iff $(\neg \dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2})$ or $(\dot{c} \wedge \dot{C}_1 \wedge \ldots \wedge \dot{C}_p \wedge \texttt{w}(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C}_1 \wedge \ldots \wedge \ddot{C}_p))$ holds, where c is a primitive constraint, $\dot{c} = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, c)$; $\boldsymbol{V_0}, \boldsymbol{V_1}$ and $\boldsymbol{V_2}$ are three vector of FD_variables, $\boldsymbol{V_3}$ is a newly created vector of FD_variables, $\dot{C}_1 = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, C_1), \ldots, \dot{C}_p = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, C_p)$, and $\ddot{C}_1 = subs(\boldsymbol{V_1} \leftarrow \boldsymbol{V_3}, C_1), \ldots, \dot{C}_p = subs(\boldsymbol{V_1} \leftarrow \boldsymbol{V_3}, C_p)$; *subs* being the substitution of variables over a term.

The operational semantic of `w/5` is defined by the following rules :

Definition 7 `w/5` *(operational semantic)*
$\texttt{w}(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \ldots \wedge C_p)$ *is reduced to the four following guarded–constraints :*

- $\dot{c} \longrightarrow (\dot{C}_1 \wedge \ldots \dot{C}_p \wedge w(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C}_1 \wedge \ldots \wedge \ddot{C}_p))$
- $\neg \dot{c} \longrightarrow \boldsymbol{V_0} = \boldsymbol{V_2}$
- $\neg(\dot{c} \wedge \dot{C}_1 \wedge \ldots \dot{C}_p) \longrightarrow (\neg \dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2})$
- $\neg(\neg \dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2}) \longrightarrow (\dot{c} \wedge \dot{C}_1 \wedge \ldots \dot{C}_p \wedge w(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C}_1 \wedge \ldots \wedge \ddot{C}_p))$

The first two guarded–constraints result from the behavior of the *while* statement. Whenever the decision of the statement $\dot{c}$ is verified, then the body is executed and another `w/5` is stated. When the decision is refuted, the body is skipped and the input variables of the statement are equated to the vector of used variables.
The third guarded–constraint is based on the following observation : if the constraints of the body are inconsistent w.r.t the current information in the store, then the loop cannot be performed. The last guarded–constraint comes from the following observation : if the value of a variable is different before and after the *while* statement, then the body of the loop must be executed at least once. Note that the guards of both combinators are either primitive constraints or negations of conjunction of constraints, so the implementation of *abs–entailment* becomes straightforward (see section 4).
Let us illustrate how `w/5` works on the example of Fig. 2. The *while_do* statement is translated into : $\texttt{w}(J_3 * U_3 \leq 16, [J_1], [J_2], [J_3], J_2 = J_3 + 1)$. If the store contains $J_1 = 1, J_3 = U_3$, then the fourth guarded–constraint is activated because $\neg(\neg(J_1 * U_3 \leq 16) \wedge J_1 = J_3)$ is entailed by the store. So, the following constraints are added to the store : $J_1 * U_3 \leq 16 \wedge J_2 = J_1 + 1 \wedge \texttt{w}(J_3 * U_3 \leq 16, [J_2], [J_\#], [J_3], J_\# = J_3 + 1)$ where $J_\#$ is a newly created variable.

Procedure Call. A procedure call is translated into a goal to solve. For example, a statement such as $v = foo(x, 29)$ is translated into
$\texttt{foo}([X, 29], [], Liste\_of\_locals, [], V)$, where `foo` is the name of the clause generated for the procedure foo and *Liste_of_locals* is a the list of FD_variables associated to local variables and referenced in the decisions of the procedure. Such a mechanism allows the treatment of recursive procedure.

3.3 Generation of the CLP Goal

The decisions which must be verified to reach a given point in a procedure are called the *control-dependencies* [FOW87]. They are syntactically determined in well–structured procedures. For loop statements, these decisions are computed dynamically. Let $C(foo, 10)$ be the *control-dependencies* associated with point 10 in the procedure foo of Fig. 1. So, we have : $C(foo, 10) = (Z \leq 8) \wedge (U_3 \leq X) \wedge (T_2 \leq 20)$. The selected point determines a goal to solve with the clauses of the generated CLP(FD) program :

$$\longleftarrow C(foo, 10), foo([X, Y], [], [X, Z_1, U_3, T_2], [], RET)$$

The generated CLP program for program `foo` and the goal associated with point 10 are given in the Fig. 3.

$$
\begin{array}{l}
\text{foo}([X,Y],[],[X,Z_1,U_3,T_2],[],RET) \longleftarrow \\
\quad X,Y,Z,T_1,T_2,U_1,U_2,U_3,RET \in 0..2^{32}-1, \\
\quad Z = X * Y, \\
\quad T_1 = 2 * X, \\
\quad \texttt{ite}(X < 4, U_1 = 10 \wedge U_3 = U_1, U_2 = 2 \wedge U_3 = U_2), \\
\quad \texttt{ite}(Z \leq 8, \\
\quad\quad \texttt{ite}(U_3 \leq X, T_2 = T_1 - Y \wedge \\
\quad\quad\quad \texttt{ite}(T_2 \leq 20, \\
\quad\quad\quad\quad \ldots \\
\quad RET = \ldots \\
\\
\longleftarrow (Z \leq 8) \wedge (U_3 \leq X) \wedge (T_2 \leq 20), foo([X,Y],[],[X,Z_1,U_3,T_2],[],RET)
\end{array}
$$

Fig. 3. CLP Program generated for the program foo

4 Solving the Goal

In our framework, the constraint solving process is based on :

1. a filtering process based on partial consistency techniques and entailment techniques ;
2. a search procedure which combines an enumeration process and a constraint propagation step.

In view of the operational semantics of combinators introduced in the previous section, there are several operations to be implemented. They include an entailment test, an algorithm for processing the guarded–constraints, and the implementation of the combinators themselves.

4.1 Entailment Test

Three levels of entailment relaxations may be used to achieve entailment tests : *domain–entailment*, *interval–entailment* and *abs–entailment*, defined in section 2.

Consider the following example : $\sigma = (X \in 1..100) \wedge (Y \in 9..11) \wedge (X \neq Y)$ and the question " is $(X * Y \neq 100)$ entailed by σ ?".
The constraint is neither interval-entailed, nor domain-entailed because $(X = 10, Y = 10)$ does not verify the constraint. Thus, in our framework, we have implemented *abs–entailment* which is more effective —at least on our problems— than *domain-entailment* and *interval-entailment*. Practically, we add the negation of the considered constraint C to the store before starting a filtering step by interval–consistency. When the domain of one variable is reduced to an empty set, constraint C is entailed ; when all the constraints are interval–consistent, no deduction can be done and the previous store must be restored. For instance, filtering the store $\sigma \wedge \neg C = (x \in 1..100) \wedge (y \in 9..11) \wedge (x \neq y) \wedge (x * y = 100)$ by interval-consistency leads to an empty domain for both variables, and then proves that the constraint $x * y \neq 100$ is *abs-entailed*.

This relaxation of entailment can be seen as a proof by refutation. Technically, *abs-entailment* requires to compute the negation of the considered constraint C. Since we only test the entailment of primitive constraints or the negation of conjunctions of constraints in our framework, this computation becomes straightforward.

Note also that no suspension will remain in the constraint store at the end of the resolution, since the last step of the solving process is an enumeration step.

4.2 Processing Guarded–Constraints

The guarded–constraints are evaluated iteratively in the store. The algorithm for processing guarded–constraints is given in Fig. 4.

/* Let C_1, C_2 be two constraints and σ be the current store */
/* Process $C_1 \longrightarrow C_2$ in σ */

if filtering $\sigma \wedge \neg C_1$ by interval–consistency yields an inconsistency
then /* C_1 is *abs–entailed* by σ */

$\sigma \leftarrow (\sigma \cup \{C_2\}) \setminus \{C_1 \longrightarrow C_2\}$;

if filtering $\sigma \wedge C_1$ by interval–consistency yields an inconsistency
then /* $\neg C_1$ is abs–entailed by σ */

$\sigma \leftarrow \sigma \setminus \{C_1 \longrightarrow C_2\}$;

if neither C_1 nor $\neg C_1$ are abs–entailed by σ
then continue
/* The guarded–constraint $C_1 \longrightarrow C_2$ is suspended in σ */

Fig. 4. Algorithm for processing guarded–constraints

Note that the second rule can be ignored until the end of the computation because it does not add any constraint to the store. Two kind of problems may occur with this algorithm :

- the store may contain other guarded–constraints which are activated as soon as a filtering is started ;
- the store may contain a non–terminating combinator. In fact, some `w/5` combinator may introduce guarded–constraints which will recursively put other `w/5` combinators in the store. This pitfall can be seen as a consequence of the halting problem.

A practical solution for both difficulties consists in ignoring any other guarded–constraint or combinator of the store during the filtering of $\sigma \wedge \neg C_1$. Other awakening policies exist[Got00] but are not discussed in this paper.

4.3 Search Process

Filtering by partial consistencies does not always yield a solution, thus a search step is necessary. Note that, up to this point, no choice point has been set up. In fact, the disjunctions introduced by the combinators are "captured" by the entailment tests. As usual, the search is interleaved with constraint propagation. Since the class of programs is unbound, experiments are the best way to determine a good heuristic for the ATDG problem. We have tested the *first-fail*, *first-fail constrained*, *domain-splitting* heuristics among others. Iterative domain–splitting yields the best results in average [Got00].

The search process stops in one of the following states:

- **Success : a solution of the constraint system was found.** In our framework, such a solution is a test datum on which the selected point n is reached in the procedure P, hence it is a solution to the ATDG problem.
- **Success : the inconsistency of the constraint system has been detected.** If an inconsistency of the store is detected during the initial filtering step or during the search process, we can state that n is unreachable in P, i.e. there is no test datum on which n is executed[6] ; hence, the ATDG problem has no solution. This is an important information for the tester.
- **Failure : the search process did not reach a success state during the allowed amount of CPU time.** This can result from the non–termination problem of `w/5`. Consider a reachable point n in a procedure containing a loop which does not terminate for certain input values. If such an input is tried during the search process, the `w/5` combinator will not terminate.
  Note that no information can be deduced when the process is stopped before the end. It is not possible to determine whether it is a consequence of an infinite loop or just a very long search. In both cases, we say that our technique fails to find a solution of the ATDG problem.

5 First Experimental Results

We compare our CLP framework with a random test data generation method and the dynamic approach of [Kor90,FK96]. We implemented the random method by using the `drand48` C function, which generates pseudo-random numbers with the well-known linear congruential algorithm and 48-bit integer arithmetic. TESTGEN is an implementation of the dynamic method for Pascal programs. The tool is not available, hence we base the comparison on the results published in [FK96]. The symbolic execution method has been implemented in a tool called GODZILLA [DO93] but the tool is dedicated to mutation analysis of Fortran programs making the experimental comparison very difficult.

[6] sometimes called dead code

5.1 Our Prototype System

INKA operates on a restricted subset of the C language. Unstructured statements such as *goto* statement are not handled in our framework. Pointer arithmetic, dynamic allocated structures, pointer functions, type casting, involve difficult problems to solve. Pointers are only partially supported by INKA (see section 6). Although, floating point numbers are finite in essence, they introduce problems[7] which cannot be solved within the framework introduced here. All the types of integer variables (char,short, long,...) and almost all the C operators (34 out of 42) are handled (by capturing their behavior into user–defined constraints).

INKA includes a C parser, a SSA form generator and a Constraint system producer over the `clp(fd)` library of Sicstus Prolog.

5.2 Experiments

We only present our experiments on three classical academic programs of the Software Testing Community and one real–world program but INKA has been used successfully on several other programs [Got00]. The academic programs[8] are 1) "bsearch" [DO93] which is a binary search in a sorted array ; 2) a program published in [FK96] named "sample" which contains arrays, loops and a lot of dependencies ; 3) the famous program "trityp" [DO93] which contains numerous non-feasible paths.

Finally, we introduce the results for a real–world program extracted from an avionic project, named "ardeta03". This program mainly contains complex C structures and bitwise operations but does not contain loops.

5.3 Test Procedure

For each program, a test datum for each basic block (sequence of statements without branching) is generated. Of course, this approach is not optimal to reach a complete block coverage since no coverage information is reused between two generations.

[7] The evaluation process of an arithmetical expression in a CLP system and the evaluation of the same expression in the operational software may yield different results

[8] The source code of these programs are available at http://www.essi.fr/~rueher/trityp.htm

For each selected block, we have compared INKA to the random method, and to the published results of TESTGEN. We have performed our experiments on a 300Mhz Sun UltraSparc 5. A time-out of 10 seconds per block was set. In 10 seconds, the random method generated approximatively 10^5 test data, while INKA generated only one test datum. To limit the factor of "bad luck" which may occur with the random method, we repeated 10 times the generation with different initial values for the linear congruential algorithm, and we only considered the best results.

[FK96] introduces the results of TESTGEN on the three academic programs among others. The TESTGEN technique starts with a random generation of value which determines the success of the method. They performed their experiments on a PC with 60Mhz–Pentium processor. A time–out was set to 5 minutes and the same test procedure as ours was applied, except that they repeated 10 times their search for each block. Their "coverage represents the percentage of nodes for which at least one try was successful in finding input data" [FK96]. According to this definition, they found 100% for each program.

5.4 Results

The results are shown in Fig. 5. The number of lines of code and the number of statement blocks are reported in the first two columns ; whereas an estimate of the search space is reported in the third column (number of possible test data). The last three columns contain the results of block coverage obtained with the three different approaches.

Programs	loc	blocks	test data	TESTGEN*	Random**	INKA**
bsearch	21	10	$> 10^{50}$	100%	100%	100%
sample	33	14	$> 10^{100}$	100%	93%	100%
trityp	40	22	$> 10^{10}$	100%	86%	100%
ardeta03	157	38	$> 10^{60}$	–	74%	100%

(*) 50 minutes on PC Pentium (60Mhz) for each block
(**) 10 seconds on Sun Sparc 5 (300Mhz) under Solaris 2.5 for each block

Fig. 5. Comparison on block coverage

5.5 Analysis

TESTGEN did allow 50 minutes per block whereas INKA did not spent more than 10 seconds on each block. The tests with TESTGEN have been done on a PC with 60Mhz-Pentium processor while INKA was run on 300Mhz Sun UltraSparc 5. If we assume that there is less than a factor 30 between these two computers, INKA is still 10 time faster than TESTGEN[9].

[9] Note that INKA is written in Prolog while TESTGEN is written in C

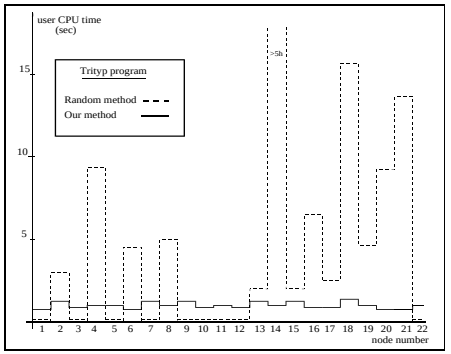


Fig. 6. Time required to generate a solution for each block

Let us see in more details what appends on one of the programs. We report in Fig. 6 the curve of times required to generate a solution for the program "trityp" by the last two methods. First, note that the time required by the random method is smaller on some blocks. In fact, INKA requires a nominal time to generate the constraint system and to solve it, even if it is very easy to solve. Second, note that the random method fails on some blocks. For instance, the block 14 which requires for the random method to generate a sequence of three equal integers. On the contrary, this block does not introduce a particular difficulty for INKA, because such a constraint is easily propagated.

6 Perspective

First experiments are promising but, of course, more experiments have to be performed on non-academic programs to validate the proposed approach. The main extension of our CLP framework concerns the handling of pointer variables. Unlike scalars, pointer variables cannot directly be transformed into logical variables because of the aliasing problem. In fact, an undirect reference and a variable may refer to the same memory location at some program point. In [Got00], we proposed to handle this problem for a restricted class of pointers : pointers to stack–allocated variables. Our approach, based on a pointer analysis, does not handle dynamically allocated structures. For some classes of applications, this restriction is not important. However, the treatment of all pointer variables is essential to extend our CLP framework to a wide spread of real–world applications.

Acknowledgements

Patrick Taillibert and Serge Varennes gave us invaluable help on the work presented in this paper. Thanks to François Delobel for its comments on an earlier draft of this paper. This research is part of the Systems and Software Tools Support department of THOMSON-CSF DETEXIS.

References

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[DO93] R. A. Demillo and A. J. Offut. Experimental Results from an Automatic Test Case Generator. *Transactions on Software Engineering Methodology*, 2(2):109–175, 1993.

[FK96] Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation". *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and J. David Warren. The Program Dependence Graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9-3:319–349, July 1987.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proc. of the Sigsoft International Symposium on Software Testing and Analysis*, Clearwater Beach, Florida, USA, March 2-5 1998. *Software Engineering Notes*,23(2):53-62. available at http://www.essi.fr/ rueher/.

[Got00] A. Gotlieb. *Automatic Test Data Generation using Constraint Logic Programming*. PhD thesis, PHD Dissertation (in French), Université de Nice–Sophia Antipolis, January 2000.

[HSD98] Pascal Van Hentenryck, Vijav Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998. Also in CS-93-02 Brown–University 1993.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kor90] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, august 1990.

[Nta98] S. Ntafos. On Random and Partition Testing. In *Proceedings of Sigsoft International Symposium on Software Testing and Analysis*, volume 23(2), pages 42–48, Clearwater Beach, FL, March,2-5 1998. ACM, SIGPLAN Notices on Software Engineering.

Cet article fait écho au lancement du projet RNTL[1] InKa (2000-2002), qui avait pour ambition de faire passer notre prototype de recherche à l'état de logiciel commercialisable. En parallèle du développement industriel de l'outil InKa, trois pistes de recherche ont été explorées dans le cadre de ce projet: le traitement des pointeurs et de la synonimie, sujet que nous présentons dans le chapitre 5, le traitement des calculs sur les nombres à virgule flottante qui connaît des développements importants sur lesquels nous reviendrons dans le chapitre 7, et le traitement efficace des appels de fonctions qui est évoqué implicitement dans l'article du chapitre 9. Le projet InKa a également connu une suite, le pojet RNTL DANOCOPS (2004-2006), qui a abordé la problématique plus générale de l'utilisation des contraintes en vérification formelle de programmes.

Les projets InKa (2000-2002) et DANOCOPS (2004-2006), ayant regroupés des équipes de recherche de Thales, du LIFC à Besançon, du CEA à Saclay, de l'I3S à Nice Sophia Antipolis, du LIG à Grenoble, ont permis de structurer un groupe de recherche actif en France autour de la thématique du test à base de contraintes. Plusieurs initiatives ont vu le jour grâce à ce groupe de recherche ; en particulier, les projets ACI V3F (2003-2006) et SESUR CAVERN (2007-2011) sur lesquels nous reviendrons plus loin.

[1] Réseau National des Technologies Logicielles

Chapter 3

Contraintes et abstractions

Ces dernières années ont connu une véritable explosion de résultats quant à la vérification automatique de programmes. Les approches se sont multipliées et plusieurs outils dédiés aux langages utilisés dans les systèmes critiques, se sont révélés capables de passer à l'échelle. Nous évoquons ici quelques pistes récentes ayant connus des développements spectaculaires, de sorte à positionner les trois articles que nous avons choisis pour illustrer ce chapitre.

Contexte: vérification formelle de programmes

La vérification de programme au niveau du code source est basée sur l'utilisation d'assertions, d'invariants, de pré/post conditions qui aident les programmeurs à spécifier des propriétés. JML (*Java Markup Language*) [Burdy 05] et Alloy [Jackson 00] sont deux languages qui ont été proposés pour écrire de telles spécifications pour Java, Spec# [Barnett 11] a été proposé pour C#, tandis que ACSL (*Ansi-C Specification Langage*) [Baudin 09] est dédié aux programmes C. Les assertions et post-conditions peuvent être contrôlées à l'exécution, mais dans certains cas, cela s'avère bien trop tardif. Par exemple, une assertion dans un programme pilotant les commandes de vol d'un avion doit impérativement être vérifiée avant exécution. De plus, la vérification à l'exécution des assertions ou post-conditions compromet le contrôle du temps d'exécution, ce qui est inadapté pour certains systèmes temps-réels. Heureusement, il existe des techniques permettant de vérifier statiquement les assertions, c'est à dire avant exécution des programmes.

On peut d'abord évoquer les outils qui s'appuient sur l'utilisation d'assistants de preuve ou de démonstrateurs automatiques. ESC/Java (*Extended Static Checker for Java*) [Flanagan 02] est un vérificateur statique de propriétés JML, telles que l'absence de division par zéro, ou bien le maintient des indices d'un tableau dans ses bornes. Cet outil repose sur l'utilisation du démonstrateur Simplify [Detlefs 05]. CAVEAT [Antoine 94] et la plateforme Why [Filliâtre 04, Bobot 11] sont des outils qui peuvent aider le programmeur à certifier des assertions ou post-conditions pour les programmes C. Spec# [Barnett 11] est un langage et un outil de vérification

formelle qui s'appuie sur l'utilisation du résolveur Z3 [De Moura 08b]. Ces outils, qui connaissent actuellement des développements importants [Barnett 11], souffrent néanmoins de certaines limitations. Souvent leur axiomatique de base est insuffisante pour obtenir une démonstration complètement automatisée, et l'intervention de l'utilisateur est nécessaire pour obtenir la preuve de certains lemmes. En ce qui concerne le traitement des boucles, ces outils réclament la donnée d'invariants de boucles qui sont difficiles à spécifier. De plus, ces approches reposent sur des hypothèses quant à la sémantique opérationnelle des langages de programmation, ce qui limite la portée des preuves obtenues.

Une autre approche classique en vérification de programme est l'utilisation de techniques d'abstractions pour inférer et contrôler des propriétés, en analysant statiquement le programme. L'Interprétation Abstraite [Cousot 77, Cousot 92] infère des propriétés sur les domaines abstraits dans le but de prouver l'absence de certaines erreurs du programme lors de son exécution (e.g., absence de débordement de capacités, absence de déréférencement de pointeurs nuls). L'analyseur statique ASTREE [Cousot 05], l'analyseur Polyspace C Verifier et l'analyseur de valeurs de Frama-C [Canet 09] sont des outils phares de l'Interprétation Abstraite, permettant de vérifier statiquement des propriétés sur des programmes C critiques. Une des limitations de l'Interprétation Abstraite provient de la difficulté de choisir le domaine abstrait adapté à la propriété à vérifier. Par contre, les approches à base d'Interprétation Abstraite sont capables de traiter automatiquement les boucles sans annotations supplémentaires, grâce à l'usage d'opérateurs particuliers (i.e., "widening/narrowing").

Egalement basé sur des techniques d'abstractions, la vérification symbolique de modèle avec abstractions[1] explore les chemins d'un modèle du programme dans le but de trouver des contre-exemples à une propriété à vérifier. Grâce à des des méthodes de raffinement automatique de modèle, SLAM [Ball 01, Ball 11] et BLAST [Henzinger 03] sont deux outils représentatifs de ce domaine de recherche qui ont connus des succès très importants en matière de recherche de contre-exemples pour les programmes C. Cependant, ces outils s'appuient des abstractions qui surapproximent l'ensemble des états atteignables et explorent des chemins de taille bornée sur les modèles. Ainsi, ces outils peuvent produire de fausses alarmes et certaines propriétés peuvent ne pas être vérifiées de par l'imprécision de la surapproximation.

[1] En Anglais, "symbolic model-checking with predicate abstraction" ou "counter-example guided abstraction refinment"

Ces dernières années, beaucoup d'attention a également été portée à l'utilisation de la résolution de contraintes pour automatiser tout ou partie de la vérification des logiciels. En 2000, Andreas Podelski soulignait déjà que la vérification formelle de programmes pouvait être vue comme une instanciation de la résolution de contraintes [Podelski 00] et proposait avec Giorgio Delzanno des techniques de "model-checking" basée sur les contraintes [Delzanno 01]. Cormac Flanagan a également proposé des fondations théoriques à l'interprétation à contraintes des programmes impératifs dans [Flanagan 04]. Dans ces approches, la vérification de programme se réduit au problème consistant à montrer qu'un système de contraintes est satisfiable ou insatisfiable. Par exemple, montrer qu'une propriété est vérifiée en un point particulier du code source conduit à résoudre un système de contraintes qui caractérise l'état du programme sur un ou plusieurs chemins qui atteignent ce point. Des travaux récents se focalisent sur l'utilisation de résolveurs de contraintes dans le contexte du "software model checking", c'est à dire du contrôle de propriétés sur des modèles bâtis par analyse du code source. Mais, c'est principalement dans le domaine du test logiciel que les outils basé sur la résolution de contraintes ont atteint un certain niveau de maturité [Godefroid 08a].

Comme évoqué dans les chapitres précédents, l'approche que nous avons proposée dès 1998 a conduit au développement de plusieurs prototypes de recherche. Les outils INKA [Gotlieb 00b, Gotlieb 06b], TAUPO [Denmat 07b], et EUCLIDE [Gotlieb 09a] sont des générateurs automatiques de test basé sur les résolveurs de contraintes SICSTUS clpfd [Carlsson 97] et clpq [Holzbaur 95], qui ont été expérimentés de manière approfondie sur des programmes C critiques. Aux Etats-Unis, l'utilisation de résolveurs SAT et de résolveurs de Programmation Linéaire pour la génération automatique de cas de test a connu des développements spectaculaires. Les travaux de Patrice Godefroid et de Koushik Sen [Godefroid 05, Sen 05] ont initié le développement de plusieurs générateurs de tests basés sur l'exécution symbolique dynamique, tels que PEX [Tillmann 08] et SAGE [Godefroid 08b] chez Microsoft, CREST [Burnim 08] à l'Université de Berkeley, ou EXE [Cadar 06] à l'Université de Stanford. Tirant partie des progrès importants réalisés dans le domaine des résolveurs SMT (*Satisfiability Modulo Theory*), certains de ces outils ont démontré des capacités impressionnantes de passage à l'échelle [Godefroid 09]. En France, Bruno Marre et Nicky Williams ont développé au CEA le générateur de tests GATEL [Marre 00] pour les programmes Lustre, et PathCrawler [Williams 05] pour les programmes C. Ces outils, qui s'appuient sur Colibri, un résolveur de contraintes basé sur la propagation de contraintes, sont utilisés pour générer des tests dans des applications de taille industrielle. Hélène Collavizza, Michel Rueher et Pascal Van Hentenryck ont proposé CPBPV pour la vérification de propriétés sur les programmes Java [Collavizza 08]. CPBPV s'appuie sur l'utilisation de deux résolveurs de contraintes, i.e., ILOG CPLEX et JSolveur [Leconte 06, Berstel 10], et se montre compétitif avec certains "model-checkers" dédiés à l'analyse et à la vérification de code source.

Les articles sélectionnés

Les trois articles présentés dans ce chapitre introduisent une approche originale pour le test à base de contraintes : l'utilisation de techniques de calculs sur les domaines abstraits pour améliorer la résolution de contraintes. Afin d'obtenir une résolution efficace des systèmes de contraintes issus des modèles, nous avons bâti une procédure qui utilise les domaines abstraits tels qu'ils ont été proposés en Interprétation Abstraite [Cousot 77, Cousot 92]. A l'instar d'autres approches telles que celle suivie dans le résolveur Abscon [Merchez 01] ou celle de [Truchet 10], nous avons montré l'existence de liens étroits qui unissent les notions classiques de filtrage par consistance locale et calculs sur les domaines abstraits [Gotlieb 09d]. Nous avons également montré comment utiliser certains opérateurs d'union abstraite dans le cadre de la génération automatique de tests [Denmat 07b]. Notre travail dans ce domaine a porté sur la réalisation du logiciel Euclide, qui combine le filtrage par consistance de bornes avec une consistance à base de calculs polyédriques, pour vérifier automatiquement des propriétés pour les programmes C critiques.

A. Gotlieb. *EUCLIDE: A constraint-based testing platform for critical c programs*. **In 2th IEEE International Conference on Software Testing, Validation and Verification (ICST'09), Denver, CO, Apr. 2009.**

Euclide: A Constraint-Based Testing framework for critical C programs*

Arnaud Gotlieb
INRIA Rennes - Bretagne Atlantique
Campus Beaulieu, 35042 Rennes Cedex, France
Arnaud.Gotlieb@irisa.fr

Abstract

Euclide is a new Constraint-Based Testing tool for verifying safety-critical C programs. By using a mixture of symbolic and numerical analyses (namely static single assignment form, constraint propagation, integer linear relaxation and search-based test data generation), it addresses three distinct applications in a single framework: structural test data generation, counter-example generation and partial program proving. This paper presents the main capabilities of the tool and relates an experience we had when verifying safety properties for a well-known critical C component of the TCAS (Traffic Collision Avoidance System). Thanks to Euclide, we found an unrevealed counter-example to a given anti-collision property.

1 Introduction

Context. Safety-critical systems must be thoroughly verified before being exploited in commercial applications. In these systems, software is often considered as the weakest node of the chain and many efforts are deployed in order to reach a satisfactory testing level. A challenge in this area is the automation of the test data generation process for satisfying functional and structural testing requirements. For example, the standard document which currently governs the development and verification process of software in airborne system (DO-178B) requires the coverage of all the statements, all the decisions and MC/DC at the highest level of criticality and it is well-known that DO-178B structural coverage is a primary cost driver on avionics project.

In addition, the verification process of critical systems often requires the verification of safety properties, as people's life may rely on these properties. For airborne systems, some safety properties can be extracted from specification documents that describe the so-called anti-collision theory regulating the controlled airspace. Checking these safety properties is mandatory and is usually preformed by manual code reviews. Although they are widely used, most of the existing testing tools on the market are currently restricted to test coverage monitoring and measurements. Coverage monitoring answers to the question: what are the statements or branches covered by the test suite ? while coverage measurements answers to: how many statements or branches have been covered ? But these tools usually cannot find the test data that can execute a given statement, branch or path in the source code. In most industrial projects, the generation of structural test data is still performed manually and finding automatic methods for this problem remains a holly grail for most testers. Nevertheless, several experimental tools exist for C programs including INKA [20], PATHCRAWLER [32, 27], CUTE [30] or PEX [31], but none of them can also check safety properties or generate counter-examples that invalidate safety properties. Software model-checking tools such as SAVE [9], MAGIC [6], BLAST [23] or CBMC [8] have been proposed for checking properties over a piece of C code. But, these tools usually cannot generate a test suite that covers selected structural criteria. Finally, proof-based environments such as WHY/CADUCEUS [18] can automatically prove properties for C programs. But these tools cannot generate test cases or counter-examples.

Euclide. In this paper, we propose Euclide a constraint-based testing tool that features three main applications: structural test data generation, counter-example generation and partial program proving for critical C programs. The core algorithm of the tool takes as input a C program and a point to reach somewhere in the code. As a result, it outcomes either a test datum that reaches the selected point, or an "unreachable" indication showing that the selected point is unreachable. Optionally, the tool takes as input additional safety properties that can be given under the form of pre/post conditions or assertions directly written in the code. In this case, Euclide can either prove that these properties or assertions are verified or find a counter-example when there is one. As these problems are undecidable in the general case, Euclide only provides a semi-correct procedure (when it terminates, it provides the right answer) for them. Hopefully, by restricting the subset of C that the tool can handle (no dynamic memory allocation, no recursion) these non-termination problems remain infrequent in practice. In addition, Euclide implements several procedures that combine atomic calls to the core algorithm. For example, by selecting appropriate points to reach in the source code, the tool can generate a complete test suite able to cover the all_statements or the all_decisions criteria.

Providing a tool able to deal with these three applications (structural test data generation, counter-example generation and partial program proving) in a single framework offers several advantages:

- For the developers having to maintain code they did not wrote, using a tool able to generate a failure-causing test datum that reaches a given point facilitates the debugging process. In fact, the test datum can easily be submitted as input to a symbolic debugger that will drive the computation towards the failure-causing point in the code ;
- In the unit testing phase, achieving high coverage with a test set that satisfies safety assertions improves the quality of the test selection process. The issued test set favorably enriches the set of tests to replay for future versions of the software (Regression Testing) ;
- For certification purposes, it is convenient to work only on a single certification product, namely the source code along with its annotations (assertions and pre/post conditions). Showing that the program satisfies all the required safety properties and that all parts of the program are executable and have been tested with respect to these properties is certainly a good way to convince a certification authority that the developed software is correct and reliable.

The underlying technology of Euclide is Constraint-Based Testing (CBT). Constraint-Based Testing is a two-stage process consisting first to generate a constraint system that corresponds to the testing objective we want to reach (for example, a selected point in a source code) and then, second to solve the constraint system by using well-recognized constraint programming techniques. CBT received considerable attention these latter years as constraint programming emerged as a worthwhile programming paradigm and solving techniques have been much improved.

Contributions. The originality of Euclide comes from its unique way of combining symbolic and numerical analyses such as static single assignment form, constraint propagation, integer linear relaxation and search-based test data generation. Static single assignment form (SSA) relieves the tool from using costly and path-oriented symbolic evaluation techniques for generating the constraint system. Indeed, SSA allows considering several paths going through the selected point to reach at the same time. Thanks to constraint propagation, Euclide nicely handles non-linear operations such as multiplication between unknown variables, division, conditional and loop statement within C programs. Thanks to integer linear relaxation, the tool handles efficiently linear operations over integer variables. It also detects some unsatisfiable (possibly non-linear) constraint systems which were unbearable without this technique. Finally, thanks to its search-based test data generator that cooperatively labels the variables according to distinct heuristics, Euclide can generate test data or counter-examples in very efficient way. In this paper, we do not claim that Euclide is better than other more specialized test data generators or software model-checkers, but we show that this is its combination of symbolic and numerical techniques that offer the opportunity to get results outside of the scope of other tools. We exemplify this statement by our recent experience on using Euclide to prove safety properties for a well-known critical C component of the TCAS (Traffic Collision Avoidance System). Thanks to Euclide, we found an unrevealed counter-example to a given anti-collision property.

Plan of the paper. The rest of the paper is organized as follows: Section 2 reviews the main technologies used in Euclide. Section 3 presents its architecture and implementation while Section 4 relates our experience in using Euclide for generating test data and checking safety properties of a critical module of the TCAS. Section 5 presents the related work and finally, Section 6 concludes and draws some perspectives to this work.

2 Constraint generation and solving

2.1 Critical ISO/IEC compliant C programs

Our approach is dedicated to the testing of safety-critical (and ISO/IEC compliant) C programs. These programs share some characteristics such as being written in a restricted subset of the C language that excludes recursion and dynamic memory allocation among other things. The C language, as defined by the ISO/IEC standard [33], has also the considerable advantage to be well defined in terms of syntax and semantics, even if several operations have still an undefined behavior[1] or a behavior defined by the imple-

*This work is partially supported by ANR through the RNTL CAT and the CAVERN projects under the reference ANR-07-SESUR-003

[1] Exact behavior which arises is not specified by the standard, and exactly what will happen does not have to be documented by the C implementation.

mentation (in particular for floating-point computations).

Euclide handles a subset of C that includes integer and floating-point computations, pointers towards named locations, arrays of statically-allocated size, structures, function calls, bit-to-bit operations such as masks, all control structures (including loops) and almost all operators (34 over 42). But, it also has some restrictions: it does not deal accurately with unstructured statements such as gotos, unconstrained pointer arithmetic (such as using a physical address of a memory segment or adding two unrelated addresses as if they were integers), function pointers, functions with a unknown number of parameters, volatiles, unions, memory type casting (such as reading an integer as it was an address), library and external function calls (unavailable source code).

2.2 Generating Euclide programs

Euclide is based on a constraint model of C programs. This model, expressed in a dedicated language, is extracted from the source code by several transformational passes: parsing, normalization, pointer analysis, Static Single Assignment form and constraint model generation. In this section, we briefly review all these passes and discuss the main technologies used in Euclide to generate and solve constraint systems corresponding to testing objectives.

Parsing and normalization. This pass consists in building a symbol table and an abstract syntax tree for each compilation unit (preprocessed program). The symbol table keeps track of the type, scope, memory allocation class of each variable of the program while the abstract syntax tree captures the syntax of all the (non-declarative) statements of each function. Normalization is a process that permits to break complex statement into simpler ones. The rationale behind this pass is to simplify other passes by considering a smaller set of statements to analyze. Complex control structures are rewritten into simpler ones, function calls and arguments are isolated as well as side-effect expressions, multi-operators statements are decomposed. For example, thanks to the introduction of new temporary variables, a complex assignment statement such as `e=v1*v2*f()+v3;` is decomposed into `t0=f(); t1=v1*v2; t2=t1*t0; e=t2+v3;` because the function call has a higher priority than `*` and `+` and operands are evaluated from left to right. Note that such decomposition correctly handles multi-occurrences in C expressions. In the presence of floating-point computations, special attention must be paid to preserve the semantics. In particular, the decomposition requires that intermediate results of an operation conform to the type of storage of its operands[2]. In the previous example, if `v1` and `v2` are of single-format, then the temporary variable `t1` must also be single-format. For floating-point computations, this process has been extensively presented in a dedicated paper [4].

Points-to analysis. Euclide implements a *points–to analysis* that statically collects a set of variables that may be pointed by the pointers of the program and determines the set of memory locations that can be accessed through a dereference [21]. We selected a flow-sensitive points-to analysis previously introduced by *Emami et al.* [16] where each points-to relation is a triple: $pto(p, a, definite)$ or $pto(p, a, possible)$ where a denotes a variable pointed by p. In the former case, p points definitely to a on any control flow path that reaches the statement where the pointing relation has been computed. In the latter case, p may points to a only on some control flow paths. In a flow-sensitive analysis, the order on which the statements are executed is taken into account and the analysis is computed on each statement of the program.

Single Static Assignment form (SSA). A key-feature of Euclide concerns its use of the SSA form to avoid the usual costly path exploration phase of other tools. The SSA form is a semantics-preserving transformation of a program where each variable has a unique definition and every use of this variable is reached by the definition. Performing this transformation requires to rename uses and definitions of the variables. For example `i=i+1; j=j*i` is transformed into `i2=i1+1; j2=j1*i2`. At the junction nodes of the control structures, SSA introduces special assignments called ϕ-functions, to merge several definitions of the same variable : `v3 = φ(v1,v2)` assigns the value of `v1` in `v3` if the flow comes from the first branch of the decision, the value of `v2` otherwise. SSA provides special expressions to handle arrays : `access(a,k)` which evaluates to the kth element of `a`, and `update(a0,j,v)` which evaluates to an array `a1` which has the same size and the same elements as `a0`, except for `j` where value is `v`. In the presence of pointers, special care must be taken when expliciting the possible hidden definitions of variables. We therefore defined a special form called Pointer SSA that captures hidden definitions through the usage of new special assignments exploiting the results of the flow-sensitive points-to analysis. The interested reader can consult [21] to get more details on our implementation of the so-called *Pointer SSA form* which accurately captures hidden definitions due to dereferences.

Constraint generation. Finally, statements under SSA form are converted into constraints in a dedicated intermediate language (not very inventively called Euclide). Relations, which are units of the language, can be either user-defined or primitive. User-defined relations correspond to functions defined in the C program while primitive relations are relations provided by the language itself. A relation can call other relations, allowing so to capture the C function calling mechanism. Examples of primitive relations include the ITE relation that models a conditional statement or the W relation modeling iterative statements. We will discuss the ITE relation in details in Sec.2.3 while details on W can be found in [13]. Evaluating an Euclide program yields either to true (= 1), or false (= 0) or suspend (= 0..1), corresponding to the truth value of the last evaluated relation of the program. Evaluation is incremental and relations can be awoken by additional relations. Fig.1 contains a simple Euclide program that implements a relational version of the *greatest common divisor* algorithm. Note that this Euclide program has been automatically generated from the imperative version of the gcd program.

```
rel GCD (X, Y, Z) iff            % true iff Z = gcd(X, Y)
{
    [X, Y, Z] in integers(unsigned, 32),
    X > 0, Y > 0, Z > 0,
    W(X > 0, [X, Y], [X4, Y2], [X5, Y3],
    {
        ITE (X < Y, [X, Y], [X2, Y1], [X3, Y2],
        {
            locals [X1],      % X1 is local to the current bloc
            X1 = X + Y,
            Y1 = X1 − Y,
            X2 = X1 − Y1,
        },
        { }                   % There is no Else_part
        ),
    X4 = X3 − Y2
    } )
}
```

Figure 1. The Euclide GCD program

On the request `GCD(X,Y,Z), X in 1..10, Y in 10..20, Z in 1..1000`, the constraint solvers of Euclide reduce the bounds of `Z` to `1..10`. Furthermore, if we add the relation `X=2*Y`, then Euclide automatically deduces that `Z` must be equal to `Y` to satisfy the request, which is a strong deduction usually outside the scope of other constraint solvers. In addition, the Euclide language includes a *reach* directive that is used to specify testing objectives. By inserting a *reach* directive in an Euclide program, the user unambiguously selects a location to reach within the source code and constrains the solutions of the program to satisfy this objective. For example, in the program of Fig.1, adding a *reach* directive in the Then-part of the conditional relation, permits to generate a test datum (values for `X`, `Y`) that reaches this part through an executable path. This *reach* directive is a key point of Euclide as it permits to specify various problems of reacheability, including structural test data generation and counter-example generation.

An error-free semantics. The Euclide program captures an error-free relational semantics of its corresponding C program. In other words, executions that yield errors such as dividing-by-zero or null pointer dereferencing are not considered when solutions of the Euclide program are seeked. In fact, Euclide aims at finding functional faults and not runtime errors (i.e. errors that cause exceptions at runtime). Typically, a functional fault occurs in a program P when P returns the value 3 when 2 was expected. Detecting functional faults is crucial in the context of safety-critical program verification as people's life may rely on it. Note that functional faults cannot be detected by existing static analyzers as there is no oracle in these tools. By focusing on functional faults only, our constraint model is also simpler to implement and more efficient, as it does not have to maintain spurious erroneous states.

2.3 Constraint solving

The most innovative part of Euclide concerns its constraint solving engine. As said previously, Euclide implements constraint propagation, dynamic linear relaxation and search-based test data generation in order to satisfy testing objectives. A testing objective can be either 1) to generate a test datum that passes through a *reach* directive, 2) to generate a counter-example (i.e. a complete path that invalidates a property) or 3) to prove that a given property is satisfied by all executions of the program. Both former cases correspond to find a solution of a constraint system while the latter corresponds to show that a certain constraint system is unsatisfiable. In the latter case, the proof is only partial because all the domains on which the proof holds are bounded.

Constraint Propagation (CP). Roughly speaking, CP considers each constraint in isolation as a filter for the variation domain of the constraint variables. Once a reduction is performed on the domain of a variable, CP is awaking the other constraints that hold on this variable in order to propagate the reduction. Technically, CP is incrementally introducing constraints into a propagation queue. Then, an iterative algorithm is managing each constraint one by one into this queue by filtering the domains of their inconsistent values. When the variation domain of variables is too large, filtering algorithms consider usually only the bounds of the domains for efficiency reasons: a domain $D = \{v_1, v_2, \ldots, v_{n-1}, v_n\}$ is approximated by the range $v_1..v_n$. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints that hold on this variable. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed. When selected in the propagation queue, each constraint is added into a constraint–store which memorizes all the considered constraints. The constraint–store is contradictory if the domain of at least one variable becomes empty. In this case the corresponding

[2] This property is not a requirement of IEEE-754 which is the standard that governs floating-point computations and consequently it is not always true. For example, on Intel's architectures extended formats are used by default to store intermediate results

testing objective is shown as being unsatisfiable.

Efficiency and completeness of CP. In the worst case, constraint propagation runs in $O(mn)$ where m denotes the number of constraints and n denotes the size of the largest domain. But constraint propagation alone does not guarantee satisfiability, as it just prunes the variation domains without looking at potential solutions. And it must be coupled with other mechanisms in order to find solutions or to show inconsistency[3]

Dynamic Linear Relaxations (DLRs). In [14], we introduced DLRs to relax dynamically all the constraints of an Euclide program, including the non-linear ones, within a Linear Programming framework. Linear Programming techniques such as the simplex procedure can solve huge instances of linear constraint systems very efficiently. Linear relaxation can be understood as a systematic way to over-approximate Euclide's relations by linear constraints. We integrated linear relaxations within the constraint propagation process, yielding to an optimized cooperation scheme of the constraint solving process. For control structures (conditionals, loops) we proposed specific DLRs based on case-based reasoning and abstract interpretation techniques [13]. For example, the DLR of the ITE relation uses the following principles: given an ITE relation modeling a disjunction between two subpaths (Then–part and Else–part), first try to prove that one of the two disjuncts is unsatisfiable with the rest of the constraints and, thus, replace the overall disjunction by the other disjunct. Second, when this case-based reasoning fails, compute the union of both domains as in the following example: from the disjunctive constraint $X = Y \vee X = 5$ with domains $D_X = -1000..1000, D_Y = 0..1$, one can deduce that $D_X = 0..5, D_Y = 0..1$. In Euclide, we extended the union principle with linear relations. For example, considering $X = Y + 10 \vee X = Y - 10$ with domains $D_X = D_Y = 0..20$ we deduce that $-10 \leq X - Y \leq 10$ while the above reasoning over domains would not have deduce anything new on the domains.

Test Data Generation. CP and DLRs cannot guarantee satisfiability on their own as they both computes over-approximations of the sets of solutions. Hence, it is necessary to combine these processes with a labeling step in order to exhibit a solution (a test data satisfying the testing objective or a counter-example to a given property) or to demonstrate unsatisfiability (a partial proof of the property). Such a labelling step consists in exploring the input search space. One remarkable feature of modern labelling procedures is their ability to awake constraint propagation. Once a value a is assigned to a variable v, a constraint $v = a$ is added to the constraint system and awakes other constraints holding on v. Thanks to CP, the input search space is likely to be pruned before having to enumerate all the values of the variables domain. In Euclide, we implemented and experimented several heuristics to choose the variable and the value to enumerate first. Finally, we depicted a labelling procedure that enchains several heuristics: *domain constraints*, *domain splitting*, *exhaustive search*, and *random choices*. *Domain constraints* consists in exploring subdomains of the input search space by iteratively increasing the size of the explored subdomains, while *domain splitting* consists in dividing the subdomains by propagating division constraints. For example, if $x \in 0..2^{32} - 1$ then domain splitting first adds the division constraint $x \in 0..2^{31} - 1$ which will be propagated throughout the constraint system and second it adds $x \in 2^{31}..2^{32} - 1$. *Exhaustive search* is the process that will enumerate all the values in the increasing or decreasing order of a given single dimension subdomain while *random choices* will pick up values at random within a domain. Thanks to these heuristics, search-based test data generation allows to find solutions in most cases. However, as the problem of finding solutions of a non-linear constraint system over finite domain is NP_hard [22], it may happen that the search fails in a reasonable amount of time. For these reasons, we implemented a parameterized time-out process to the search.

[3]Proving a property over a piece of code in Constraint-Based Testing requires showing that a constraint system is unsatisfiable.

3 Architecture and implementation

Euclide features three main applications: structural test data generation, counter-example generation and partial program proving. The tool architecture shown in Fig.2 and its implementation were thought with these applications in mind.

3.1 Architecture

The tool takes a set of C files as input, optionally annotated by pre/post conditions and assertions (input column). For each C function of the files, an intra-procedural control flow graph is built and can be displayed through a graphical user interface (Control flow graph generator and CFGs component of the output column). In addition, an Euclide program is generated through the passes that have been presented above (parsing, normalization, points-to analysis, SSA form, constraint generation). Selecting either a node or a branch to reach yields to add a *reach* directive within the intermediate Euclide program (testing objectives of the input column). From there, constraint solving is launched according to some parameterization through an evaluator component. When a test data is generated, the flow is monitored either on the control flow graph or on a textual view of the Euclide program. The value of each individual input is shown and recorded when agreed by the user. Optionally, the linear relations that over-approximate each intermediate state of the analysis are printed within an intermediate file. When the testing objective is unsatisfiable (non-feasible point or unsatisfiable property), then this is reported to the user. In addition, several automatic structural test data generation procedures are available such as generating a test set that covers all the executable statements or decisions. These procedures use several algorithms that add *reach* directives in appropriate locations.

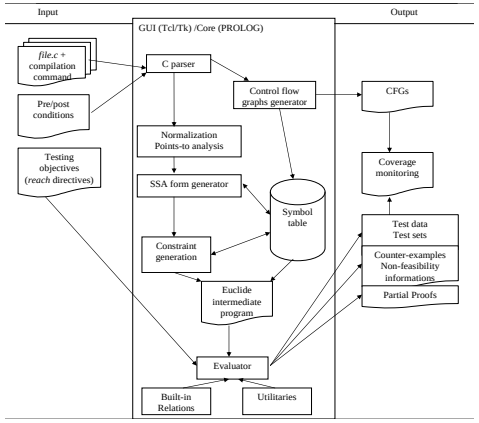


Figure 2. Euclide's architecture

3.2 Implementation

The Euclide's implementation includes 9 internal components (inside the box of Fig.2) and two additional interface components. The tool is mainly developed in Prolog (∼10 KLOC), C (∼0.3 KLOC) and Tcl/Tk (∼0.5 KLOC). The internal components include a backtrackable C parser written with the Definite Clause Grammar of Prolog, a SSA form generator based on the single-pass generator of Brandis and Mossenbock [5], an Euclide program generator and parser, a built-in relations library that implements most of the C operations (conditionals, loops, bit-to-bit operators, logical operators, function call operator, access/update, memory operations,...) and an utilitary component. The additional interface components implement the graphical user interface in Tcl/Tk and the batch mode in Prolog. Floating-point low-level representation and operations are implemented in C.

The evaluator component implements several constraint solvers that make use of the two following libraries: the *clpfd* library of Sicstus Prolog which implements a finite domains constraint solver ; and the *clpq* library that implements a linear programming solver based on simplex over the rationals. We made the two solvers cooperate by implementing our own constraint propagation queue and by building a dedicated constraint propagation solver.

4 Case study

Euclide is a Constraint-Based Testing tool dedicated to the validation of critical C programs and besides the traditional validation on academic examples, we wanted to evaluate the capabilities of Euclide on a real-world program. A typical (but small) example is the well-documented TCAS component of the Siemens suite. This suite was initially provided by Thomas Ostrand and its colleagues at Siemens Corporate Research Unit for an experimental study of the fault detection capabilities of coverage criteria [24]. It was then exploited by both Industry and Academia to evaluate testing strategies. Each component of the suite comes with a set of test cases and a set of mutants that exemplify typical faults. Recently, the suite was made publicly and freely available through the Software-artifact Infrastructure Repository [15].

TCAS (Traffic Alert and Collision Avoidance System) is an on-board aircraft conflict detection and resolution system embedded on all commercial aircrafts. The system is intended to alert the pilot to the presence of nearby aircraft that pose a mid-air collision threat and to propose maneuvers so as to resolve these potential conflicts. In cases of collision threats, the TCAS enters some levels of alertness. As shown on Fig.3, when an intruder aircraft enters a protected zone, the system issues a Traffic Advisory (TA) to inform the pilot of potential threat. In addition, TCAS estimates the time remaining until the two aircrafts reach the closest point of approach (CPA). If the danger of collision increases then a Resolution Advisory (RA) is issued, providing the pilot with a proposed maneuver that is likely to solve the conflict. The RAs issued by TCAS are currently restricted to the vertical plane only (either climb or descend) and their computation depends on time-to-go to CPA, range and altitude tracks of the intruder.

Implementation. The main component (`tcas.c`), extracted from the Repository is responsible of the Resolution Advisories issuance. It is made up of 173 lines of C code and contains nested conditionals, logical operators, type definitions, macros and function calls. Fig.4 shows the call graph of the program while Fig.5 shows the code of the highest-level function `Alt_sep_test` which computes the RAs. This function takes 14 global variables as input, including `Own_Tracked_Alt` the altitude of the TCAS equipped airplane, `Other_Tracked_Alt` the altitude of the "threat", `Positive_RA_Alt_Thresh` an adequate separation threshold, `Up_Separation` the estimated separation altitude resulting from an upward maneu-

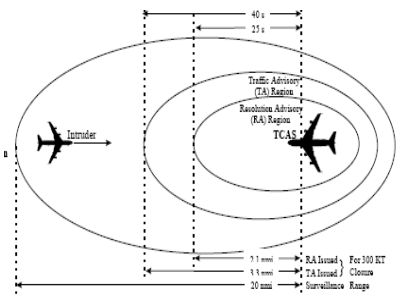


Figure 3. TCAS alarms

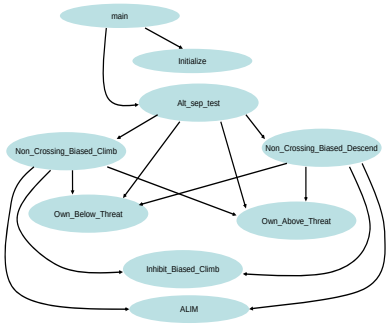


Figure 4. Call graph of `tcas.c`

```
    int alt_sep_test()
    {
1.  bool enabled, tcas_equipped, intent_not_known;
2.  bool need_upward_RA, need_downward_RA;
3.  int alt_sep;

4.  enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
5.  tcas_equipped = Other_Capability == TCAS_TA;
6.  intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;

7.  alt_sep = UNRESOLVED;

8.  if (enabled && ((tcas_equipped && intent_not_known) —— !tcas_equipped))
    {
9.  need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
10. need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
11. if (need_upward_RA && need_downward_RA)
    /* unreachable: requires Own_Below_Threat and Own_Above_Threat
    to both be true*
12. alt_sep = UNRESOLVED;
13. else if (need_upward_RA)
14. alt_sep = UPWARD_RA;
15. else if (need_downward_RA)
16. alt_sep = DOWNWARD_RA;
17. else alt_sep = UNRESOLVED;
    }

18. return alt_sep;
    }
```

Figure 5. Function alt_sep_test from `tcas.c`

ver and `Down_Separation` the estimated separation altitude resulting from a downward maneuver.

Interestingly, any TCAS implementation should be certified under level B of the DO-178B standard[4]. This has several implications w.r.t. the testing level required for certifying the TCAS. In particular, all the statements and decisions of the source code must be executed at least once during the testing process and any statement and decision must be shown as being executable, because non-executable elements do not trace to any software requirements and do not perform any required functionality.

Safety properties. In addition to these requirements, any TCAS implementation should verify safety properties that come from the aircraft anti-collision theory [28]. For the considered component, several properties referring to the possibility of issuing either an upward or a downward RA have been previously formalized [26, 9]. Tab.1 shows the five double properties extracted from [9]. For example, property P1b says that if an upward maneuver does not produce an adequate separation while an downward maneuver does, such as in Fig.6, then an upward RA should not been produced.

Results and analysis. We conducted several experiments on this program to evaluate the capabilities of Euclide to serve as an aid for certification purposes. Firstly, we evaluated structural test data generation for the coverage of the all_decisions criterion. On an Intel Core Duo 2.4GHz clocked PC with 2GB of RAM, Euclide generated a test set covering all the executable decisions of the `tcas` program in 16.9 seconds, including time spent garbage collecting, stack shifting, or in system calls. It also showed that the decision of line 11-12 of Fig.5 was non executable in less than 0.2 second. Secondly, we evaluate partial program proving on the safety properties of Tab.1. Results are shown in Tab.2. Finding counter-examples to safety properties on a TCAS implementation could appear as being dramatic. But, the reader should be warned that this TCAS implementation probably corresponds to a preliminary version and that it has probably never been used in operational conditions.

Surprisingly, we found that properties P2B, P3A and P5B were not proved w.r.t. the implementation and, thanks to Euclide, we exhibited verified counter-examples. These counter-examples satisfy the preconditions but invalidate the postconditions of the properties when submitted to the implementation. All the material of these experiments, including the test data corresponding to counter-examples, is available online[5]. In addition, counter-examples to properties P5B were not reported in the literature [9, 8, 6]. Moreover, we got these counter-examples and proofs very quickly (all the counter-examples and proofs are generated in less than 20s on our standard machine) which is encouraging for a future comparison with other more dedicated tools.

[4]The standard classifies systems under 5 criticality levels: from the highest critical level A to the least critical E

Table 1. Safety properties for tcas.c

Num.	Property	Explanation	Specifications
P1a	Safe advisory selection	An downward RA is never issued when an downward maneuver does not produce an adequate separation	`assumes Up_Separation >= Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh; ensures result != need_Downward_RA;`
P1b	Safe advisory selection	An upward RA is never issued when an upward maneuver does not produce an adequate separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation >= Positive_RA_Alt_Tresh; ensures result != need_Upward_RA;`
P2a	Best advisory selection	A downward RA is never issued when neither climb or descend maneuvers produce adequate separation and a downward maneuver produces less separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh && Down_Separation < Up_Separation; ensures result != need_Downward_RA;`
P2b	Best advisory selection	An upward RA is never issued when neither climb or descend maneuvers produce adequate separation and an upward maneuver produces less separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh && Down_Separation > Up_Separation; ensures result != need_Upward_RA;`
P3a	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	`assumes Up_Separation >= Positive_RA_Alt_Tresh && Down_Separation >= Positive_RA_Alt_Tresh && Own_Tracked_Alt > Other_Tracked_Alt; ensures result != need_Downward_RA;`
P3b	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	`assumes Up_Separation >= Positive_RA_Alt_Tresh && Down_Separation >= Positive_RA_Alt_Tresh && Own_Tracked_Alt < Other_Tracked_Alt; ensures result != need_Upward_RA;`
P4a	No crossing advisory selection	A crossing RA is never issued	`assumes Own_Tracked_Alt > Other_Tracked_Alt; ensures result != need_Downward_RA;`
P4b	No crossing advisory selection	A crossing RA is never issued	`assumes Own_Tracked_Alt < Other_Tracked_Alt; ensures result != need_Upward_RA;`
P5a	Optimal advisory selection	The RA that produces less separation is never issued	`assumes Down_Separation < Up_Separation; ensures result != need_Downward_RA;`
P5b	Optimal advisory selection	The RA that produces less separation is never issued	`assumes Down_Separation > Up_Separation; ensures result != need_Upward_RA;`

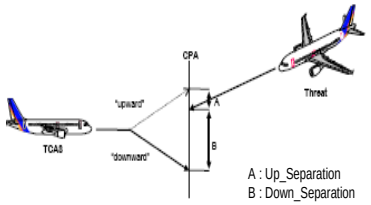


Figure 6. Resolution Advisories

Table 2. Verification of safety properties

Num	Results	Time (sec.)	Mem. (MB)
P1a	Property proved	0.7	4.6
P1b	Property proved	0.7	4.6
P2a	Property proved	0.6	4.6
P2b	Counter-example found	0.7	4.6
P3a	Counter-example found	5.4	6.3
P3b	Property proved	1.2	4.6
P4a	Counter-example found	6.8	6.9
P4b	Counter-example found	2.7	5.9
P5a	Property proved	0.6	4.6
P5b	Counter-example found	1.0	4.6

5 Related work

Euclide addresses three distinct applications for C programs, namely test data generation for structural testing, counter-example generation and partial program proving. We are not aware of any tool having the same capabilities for C programs. However, many tools exist for one or two of these tasks.

Partial program proving. These tools usually apply Floyd-Hoare logic or Dijkstra's weakest preconditions calculus to the formal verification of the so-called *verification conditions (VCs)* extracted from programs and annotations. Caduceus [18], which was pioneering deductive verification of C programs, concurrently launches several interactive proof assistants or theorem provers to prove a given assertion. Spec# [25] infers loops invariants by using abstract interpretation and infers VCs, even in the presence of dynamic allocated objects on the heap. More recently, Dash [2] exploits lightweight symbolic execution techniques and a single call to a theorem prover to show that a given property is satisfied on several paths of the implementation. Euclide implements its own automated constraint solving procedures while Caduceus, Spec# and Dash exploit existing interactive proof assistants and automated theorem provers. As a result, Euclide deals more accurately with floating-point computations [4] and more efficiently with integer-based computations as it supposes every integer variable to belong to a finite domain and implements its own dedicated constraint techniques. But Euclide is also harder to develop and is less general because its proofs are only valid for bounded integer variables.

Automatic test data generators. The test data generator Godzilla was proposed very early [12] for Fortran programs in the context of mutation testing. In a subsequent paper, the *dynamic domain reduction procedure* was developed to

[5]`www.irisa.fr/lande/gotlieb/resources.html`

enrich the constraint solving capabilities of this approach [29]. This procedure mimics the constraint propagation step described in Sec.2.3. Constraint propagation is an old idea that lates back to the beginning of the seventies and its use has been proposed very early for test data generation [3]. InKa [20] was a pioneer in the use of *Constraint Logic Programming* for generating test data for C programs. It was able to generate test case for programs containing dynamic allocated structures as its memory model was rich enough [7]. Euclide can be seen as a successor of InKa as it shares many technical features with it (both are based on SSA and Constraint Propagation). However, several distinct choices have been made for efficiency reasons. Using some apriori restrictions (no dynamic memory allocation, no recursion), the Euclide's memory model is simpler and permits to deal more efficiently with integer computations. PathCrawler [32], Dart [19] and CUTE [30] are three modern path-oriented structural test data generators. These three tools rely on path selection, symbolic execution and concolic execution. On the contrary, Euclide rely on statement or decision selection (goal-oriented approach [17]), static single assignment form and a mixture of symbolic and numeric constraint solving procedures. The treatment of loops is very different: while these path-based tools unfold the control flow structure of loops to select a path, Euclide handles a loop structure as a whole. By abstracting the behavior of the loop structure (as done with abstract interpretation techniques), the tool can deduce properties outside the scope of any path-based test data generator. For example, Euclide can (sometimes) determine that a given point, positioned after a loop structure, is unreachable. This is impossible with a path-based tool that will enumerate indefinitely all the paths through the loop structure. Recently, Bardin and Herrmann performed a remarkable work on the OSMOSE tool which aims at covering all executable paths of a binary program by using constraint solving techniques [1]. By addressing low-level binary-code, they opened a door that we could benefit from for improving the coverage of our own tool. In fact, C code often presents low-level features that we cannot currently deal with (unconstrained pointer arithmetic, dynamic jumps, ...).

Counter-example generation. Software model-checkers such as Save [9], Blast [23], Magic [6] or Cbmc [8] permit to find counter-examples to temporal properties over C programs. These tools explore the paths of a bounded model of programs in order to find a counter-example path to the property. Some of them exploit *predicate abstraction* and counter-example refinement to boost the exploration. Euclide contrasts with SAT-based or SMT-based model-checkers as it does not abstract the program and does not generate spurious counter-example paths. In particular it builds a high-level constraint model of C program by capturing an error-free semantics without considering a boolean abstraction of the program structure. Our approach has more similarities with the CPBPV tool of Collavizza, Rueher and Van Hentenryck [10, 11] that call several constraint solvers in sequence. Recently, its authors showed that CPBPV could outperform the best model-checkers on several classical benchmarks. As Euclide, CPBPV tool is based on deductive constraint programming techniques. However, research and experimental work remains to confirm these results obtained on a small set of academic programs.

6 Conclusion

In this paper, we introduced Euclide, a Constraint-based testing platform for C programs. The capabilities of the tool include structural test data generation, counter-example generation and partial program proving and it combines numerical and symbolic techniques, namely SSA, constraint propagation, dynamic linear relaxations and search-based test data generation. Euclide handles a large subset of C, even if some apriori restrictions have been done (no recursion, no dynamic allocation). The tool was applied to the verification of a critical component of the TCAS, which yields an unrevealed counter-example to a safety property. However, the tool could be improved in many ways. Function calls are currently handled by inlining which prevents Euclide from using efficient modular constraint-based analysis. Summaries of function calls could be exploited in the test data generation process. Search-based test data generation currently exploits only complete heuristics that explore the whole search space in the worst case. We could also exploit local search techniques that are sometimes very efficient. Other similar improvments are possible and requires additional research works in order to increase the efficiency of the tool.

7 Acknowledgment

Much of the choices and decisions taken within the development of Euclide were discussed with other people, and I am indebted to all of them. I would like to thanks especially Tristan Denmat who investigated the role of Abstract Interpretation in the linear relaxation techniques we employed. Many thanks also to Bernard Botella, Benjamin Cama, Florence Charreteur, Nadjib Lazaar, Bruno Marre, Matthieu Petit and Pierre Rousseau.

References

[1] Sebastien Bardin and Philippe Herrmann. Structural testing of executables. In *1th Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, pages 22–31, 2008.

[2] N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *Proc. of ISSTA'08*, pages 3–14, 2008.

[3] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. SMOTL - a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5(1):60–66, January 1979.

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

[5] M.M. Brandis and H. Mőssenbőck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Transactions on Programming Language and Systems*, 16(6):1684–1698, Nov. 1994.

[6] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[7] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *TAIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.

[8] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proc. of ASP-DAC'03*, pages 308–311, Jan. 2003.

[9] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*, pages 142–150, Vienna, Austria, September 2001. ACM.

[10] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 182–196, 2006.

[11] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.

[12] R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[13] T. Denmat, A. Gotlieb, and M. Ducasse. An abstract interpretation based combinator for modeling while loops in constraint programming. In *Proceedings of Principles and Practices of Constraint Programming (CP'07)*, Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.

[14] T. Denmat, A. Gotlieb, and M. Ducasse. Improving constraint-based testing with dynamic linear relaxations. In *18th IEEE International Symposium on Software Reliability Engineering (ISSRE' 2007)*, Trollhttan, Sweden, Nov. 2007.

[15] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[16] E. Emami, R. Ghiya, and L.J. Hendren. Context–sensitive interprocedural points–to analysis in the presence of function pointers. In *Proc. of PLDI'94*, Orlando, FL, Jun. 1994.

[17] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, Jan. 1996.

[18] J.C. Filliâtre and C. Marché. Multi-prover verification of c programs. In *6th Int. Conf. on Formal Engineering Methods (ICFEM'04)*, pages 15–29, Nov. 2004.

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223, 2005.

[20] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. of ISSTA'98*, pages 53–62, 1998.

[21] A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, Sep. 2007.

[22] P.V. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998.

[23] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of 10th Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.

[24] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE '94*, pages 191–200, 1994.

[25] Rustan Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[26] C. Livadas, J. Lygeros, and N.A. Lynch. High-level modeling and analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[27] Patricia Mouy, Bruno Marre, Nicky Williams, and Pascale Le Gall. Generation of all-paths unit test with function calls. In *First International Conference on Software Testing, Verification, and Validation, (ICST'08)*, pages 32–41, 2008.

[28] U.S. Department of transportation Federal Aviation Administration. *Introduction to TCAS II - version 7*, Nov. 2000.

[29] J.A. Offut, Z. Jin, and Pan J. The dynamic domain reduction procedure for test data generation. *Software–Practice and Experience*, 29(2):167–193, 1999.

[30] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/FSE-13*, pages 263–272. ACM Press, 2005.

[31] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proc. of ESEC/FSE-13*, pages 253–262. ACM Press, 2005.

[32] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *In Proc. Dependable Computing - EDCC'05*, pages 281–292, 2005.

[33] www.open-std.org/JTC1/SC22/WG14/www/standards. *ISO/IEC 9899 - Programming languages - C*, 1999.

Cet article et cet outil font la synthèse de notre contribution au domaine du *Test à Base de Contraintes* (CBT), même si les difficultés rencontrées et les propositions techniques sont décrites avec beaucoup plus de détails dans d'autres articles. Notre traitement des boucles s'appuie sur un combinateur original nommé `w`, qui modélise les calculs itératifs en *Programmation par Contraintes*. Durant la propagation de contraintes, les boucles sont considérées au même titre que d'autres instructions, et obéissent à une stratégie de dépliage dynamique. Lorsque le dépliage ne peut être décidé, une approche par élargissement, définie dans le cadre de l'Interprétation Abstraite, est utilisée pour propager de l'information d'un état avant itération à un état après itération. Cette contribution est résumée dans l'article suivant.

T. Denmat, A. Gotlieb, and M. Ducasse. *An abstract interpretation based combinator for modeling while loops in constraint programming*. **In Proceedings of Principles and Practices of Constraint Programming (CP'07), Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.**

An Abstract Interpretation Based Combinator for Modelling While Loops in Constraint Programming

Tristan Denmat[1], Arnaud Gotlieb[2], and Mireille Ducassé[1]

[1] IRISA/INSA
[2] IRISA/INRIA
Campus universitaire de Beaulieu 35042 Rennes Cedex, France
{denmat,gotlieb,ducasse}@irisa.fr

Abstract. We present the w constraint combinator that models while loops in Constraint Programming. Embedded in a finite domain constraint solver, it allows programmers to develop non-trivial arithmetical relations using loops, exactly as in an imperative language style. The deduction capabilities of this combinator come from abstract interpretation over the polyhedra abstract domain. This combinator has already demonstrated its utility in constraint-based verification and we argue that it also facilitates the rapid prototyping of arithmetic constraints (e.g. power, gcd or sum).

1 Introduction

A strength of Constraint Programming is to allow users to implement their own constraints. CP offers many tools to develop new constraints. Examples include the global constraint programming interface of SICStus Prolog *clp(fd)* [5], the ILOG concert technology, iterators of the GECODE system [17] or the Constraint Handling Rules [8]. In many cases, the programmer must provide propagators or filtering algorithms for its new constraints, which is often a tedious task. Recently, Beldiceanu et al. have proposed to base the design of filtering algorithms on automaton [4] or graph description [3], which are convenient ways of describing global constraints. It has been pointed out that the natural extension of these works would be to get closer to imperative programming languages [4].

In this paper, we suggest to use the generic w constraint combinator to model arithmetical relations between integer variables. This combinator provides a mechanism for prototyping new constraints without having to worry about any filtering algorithm. Its originality is to model iterative computations: it brings while loops into constraint programming following what was done for logic programming [16]. Originally, the w combinator has been introduced in [9] in the context of program testing but it was not deductive enough to be used in a more general context. In this paper, we base the generic filtering algorithm associated to this combinator on case-based reasoning and Abstract Interpretation over the polyhedra abstract domain. Thanks to these two mechanisms, w performs non-trivial deductions during constraint propagation. In many cases, this combinator can be useful for prototyping new constraints without much effort. Note that we do not expect the propagation algorithms generated for these new constraints to always be competitive with hand-written propagators.

We illustrate the w combinator on a relation that models $y = x^n$. Note that writing a program that computes x^n is trivial whereas building finite domain propagators for $y = x^n$ is not an easy task for a non-expert user of CP. Figure 1 shows an imperative program (in C syntax) that implements the computation of x^n, along with the corresponding constraint model that exploits the w combinator (in CLP(FD) syntax). In these programs, we suppose that N is positive although this is not a requirement of our approach.

```
power(X,N){                 power(X,N,Y) :-
  Y = 1;                      w([X,1,N],[Xin,Yin,Nin],[Xin,Yout,Nout],[_,Y,_],
  while(N ≥ 1){                 Nin #>= 1,
    Y = Y * X;                  [Yout #= Yin * Xin,
    N = N - 1};                  Nout #= Nin - 1]).
  return Y;}
```

Fig. 1. An imperative program for $y = x^n$ and a constraint model

It is worth noticing that the w combinator is implemented as a global constraint. As any other constraint, it will be awoken as soon as X, N or Y have their domain pruned. Moreover, thanks to its filtering algorithm, it can prune the domains of these variables. The following request shows an example where w performs remarkably well on pruning the domains of X, Y and N.

```
| ?- X in 8..12, Y in 900..1100, N in 0..10, power(X,N,Y).

N = 3, X = 10, Y = 1000
```

The w combinator has been implemented with the *clp(fd)* and *clpq* libraries of SICStus prolog. The above computation requires 20ms of CPU time on an Intel Pentium M 2GHz with 1 Gb of RAM.

Contributions. In this paper, we detail the pruning capabilities of the w combinator. We describe its filtering algorithm based on case-based reasoning and fixpoint computations over polyhedra. The keypoint of our approach is to re-interpret every constraint in the polyhedra abstract domain by using Linear Relaxation techniques. We provide a general widening algorithm to guarantee termination of the algorithm. The benefit of the w combinator is illustrated on several examples that model non-trivial arithmetical relations.

Organization. Section 2 describes the syntax and semantics of the w operator. Examples using the w operator are presented. Section 3 details the filtering algorithm associated to the combinator. It points out that approximation is crucial to obtain interesting deductions. Section 4 gives some background on abstract interpretation and linear relaxation. Section 5 shows how we integrate abstract interpretation over polyhedra into the filtering algorithm. Section 6 discusses some related work. Section 7 concludes.

2 Presentation of the w constraint combinator

This section describes the syntax and the semantics of the w combinator. Some examples enlight how the operator can be used to define simple arithmetical constraints.

2.1 Syntax

Figure 2 gives the syntax of the finite domain constraint language where the w operator is embedded.

W	$::= \mathbf{w}(Lvar, Lvar, Lvar, Lvar, Arith\_Constr, LConstr)$
If	$::= \mathbf{if}(Lvar, Arith\_Constr, LConstr, LConstr)$
$Lvar$	$::=$ var $\mid Lvar$
$LConstr$	$::= Constr \mid LConstr$
$Constr$	$::=$ var **in** int..int $\mid Arith\_Constr \mid W \mid If$
$Arith\_Constr$	$::=$ var $Op\ Expr$
Op	$::= < \mid \leq \mid > \mid \geq \mid \neq \mid =$
$Expr$	$::= Expr + Expr \mid Expr - Expr \mid Expr * Expr \mid$ var $\mid$ int

Fig. 2. syntax of the w operator

As shown on the figure, a w operator takes as parameters four lists of variables, an arithmetic constraint and a list of constraints. Let us call these parameters $Init, In, Out, End, Cond$ and Do. The $Init$ list contains logical variables representing the initial value of the variables involved in the loop. In variables are the values at iteration n. Out variables are the values at iteration $n+1$. End variables are the values when the loop is exited. Note that $Init$ and End variables are logical variables that can be constrained by other constraints. On the contrary, In and Out are local to the w combinator and do not concretely exist in the constraint store. $Cond$ is the constraint corresponding to the loop condition whereas Do is the list of constraints corresponding to the loop body. These constraints are such that $vars(Cond) \in In$ and $vars(Do) \in In \cup Out$.

Line 2 of Figure 2 presents an *if* combinator. The parameter of type $Arith\_Constr$ is the condition of the conditional structure. The two parameters of type $LConstr$ are the "then" and "else" parts of the structure. $Lvar$ is the list of variables that appear in the condition or in one of the two branches. We do not further describe this operator to focus on the w operator.

The rest of the language is a simple finite domain constraint programming language with only integer variables and arithmetic constraints.

2.2 Semantics

The solutions of a w constraint is a pair of variable lists $(Init, End)$ such that the corresponding imperative loop with input values $Init$ terminates in a state where final values are equal to End. When the loop embedded in the w combinator never terminates, the combinator has no solution and should fail. This point is discussed in the next section.

2.3 First Example: sum

Constraint `sum(S,I)`, presented on Figure 3, constrains S to be equal to the sum of the integers between 1 and I: $S = \sum_{i=1}^{n} i$

```
sum(I){
   S = 0;
   while(I > 0){
        S = S + I;
        I = I - 1;
   }
   return S;
```

```
sum(S,I) :-
   I > 0,
   w([0,I],[In,Nin],[Out,Nout],[S,_],
        Nin > 0,
        [Out = In + Nin,
         Nout = Nin - 1]).
```

Fig. 3. The sum constraint derived from the imperative code

The factorial constraint can be obtained by substituting the line `Out = In + Nin` by `Out = In * Nin` and replacing the initial value 0 by 1. Thanks to the w combinator, sum and factorial are easy to program as far as one is familiar with imperative programming. Note that translating an imperative function into a w operator can be done automatically.

2.4 Second Example: greatest common divisor (gcd)

The second example is more complicated as it uses a conditional statement in the body of the loop. The constraint `gcd(X,Y,Z)` presented on Figure 4 is derived form the Euclidian algorithm. `gcd(X,Y,Z)` is true iff Z is the greatest common divisor of X and Y.

```
gcd(X,Y){
   while(X > 0){
        if(X < Y){
             At = Y;
             Bt = X;
        }else{
             At = X;
             Bt = Y;
        }
        X = At - Bt;
        Y = Bt;
   }
   return Y;
```

```
gcd(X,Y,Z) :-
     w([X,Y],[Xin,Yin],[Xout,Yout],[_,Z],
            Xin > 0,
            [if([At,Bt,Xin,Yin],
                Xin < Yin,
                [At = Yin, Bt = Xin],
                [At = Xin, Bt = Yin]),
            Xout = At - Bt,
            Yout = Bt]).
```

Fig. 4. The gcd constraint

3 The filtering algorithm

In this section we present the filtering algorithm associated to the w operator introduced in the previous section. The first idea of this algorithm is derived from the following remark. After n iterations in the loop, either the condition is false and the loop is over, or the condition is true and the statements of the body are executed. Consequently, the filtering algorithm detailed on Figure 5 is basically a constructive disjunction algorithm. The second idea of the algorithm is to use abstract interpretation over polyhedra to over-approximate the behaviour of the loop. Function w^∞ is in charge of the computation of the over-approximation. It will be fully detailed in Section 5.3.

The filtering algorithm takes as input a constraint store ((X, C, B) where X is a set of variables, C a set of constraints and B a set of variable domains), the constraint to be inspected ($w(Init, In, Out, End, Cond, Do)$) and returns a new constraint store where information has been deduced from the w constraint. $\tilde{X}$ is the set of variables X extended with the lists of variables In and Out. $\tilde{B}$ is the set of variable domains B extended in the same way.

Input:
A constraint, $w(Init, In, Out, End, Cond, Do)$
A constraint store, (X, C, B)
Output:
An updated constraint store

w_filtering
1 $(X_{exit}, C_{exit}, B_{exit}) := propagate(\tilde{X},\ C \wedge Init = In = Out = End \wedge \neg Cond,\ \tilde{B})$
2 **if** $\emptyset \in B_{exit}$
3 **return** $(\tilde{X}, C \wedge Init = In \wedge Cond \wedge Do \wedge$
4 $w(Out, FreshIn, FreshOut, End, Cond', Do'), \tilde{B})$
5 $(X_1, C_1, B_1) := propagate(\tilde{X}, C \wedge Init = In \wedge Cond \wedge Do, \tilde{B})$
6 $(X_{loop}, C_{loop}, B_{loop}) := w^\infty(Out, FreshIn, FreshOut, End, Cond', Do', (X_1, C_1, B_1))$
7 **if** $\emptyset \in B_{loop}$
8 **return** $(\tilde{X}, C \wedge Init = In = Out = End \wedge \neg Cond, \tilde{B})$
9 $(X', C', B') := join((X_{exit}, C_{exit}, B_{exit}), (X_{loop}, C_{loop}, B_{loop}))_{Init,End}$
10 **return** $(X', C' \wedge w(Init, In, Out, End, Cond, Do), B')$

Fig. 5. The filtering algorithm of w

Line 1 posts constraints corresponding to the immediate termination of the loop and launches a propagation step on the new constraint store. As the loop terminates, the variable lists $Init, In, Out$ and End are all equal and the condition is false ($\neg Cond$). If the propagation results in a store where one variable has an empty domain (line 2), then the loop must be entered. Thus, the condition of the loop must be true and the body of the loop is executed: constraints $Cond$ and Do are posted (line 3). A new w constraint is posted (line 4), where the initial variables are the variables Out computed at this iteration, In and Out are replaced by new fresh variables ($FreshIn$ and $FreshOut$) and End variables remain the same. $Cond'$ and Do' are the constraints $Cond$ and Do where variable names In and Out have been substituted by $FreshIn$ and $FreshOut$. The initial w constraint is solved.

Line 5 posts constraints corresponding to the fact that the loop iterates one more time ($Cond$ and Do) and line 6 computes an over approximation of the rest of the iterations via the w^∞ function. If the resulting store is inconsistent (line 7), then the loop must terminate immediately (line 8). Once again, the w constraint is solved.

When none of the two propagation steps has led to empty domains, the stores computed in each case are joined (line 9). The $Init$ and End indices mean that the join is only done for the variables from these two lists. After the join, the w constraint is suspended and put into the constraint store (line 10).

We illustrate the filtering algorithm on the `power` example presented on Figure 1 and the following request:
`X in 8..12, N in 0..10, Y in 10..14, power(X,N,Y).`

At line 1, posted constraints are:
`Xin = X, Nin = N, Yin = 1, Y = Yin, Nin < 1`. This constraint store is inconsistent with the domain of `Y`. Thus, we deduce that the loop must be entered at least once. The condition constraint and loop body constraints are posted (we omit the constraints $Init = In$):
`N >= 1, Yout = 1*X, Xout = X, Nout = N-1` and another w combinator is posted:

```
w([Xout,Yout,Nout],[Xin',Yin',Nin'],[Xout',Yout',Nout'],[_,Y,_],
  Nin'>= 1,[Yout' = Yin'*Xin', Xout' = Xin', Nout' = Nin'-1]).
```

Again, line 1 of the algorithm posts the constraints `Y = Yout, Nout < 1`. This time, the store is not inconsistent. Line 5 posts the constraints
`Nout >= 1, Yout' = Yout*X, Xout' = X, Nout' = Nout - 1`, which reduces domains to `Nout in 1..9, Yout' in 64..144, Xout' in 8..12`. On line 6, w^∞`([Xout',Yout',Nout'],FreshIn,FreshOut,[_,Y,_],Cond,Do,Store)` is used to infer `Y >= 64`. `Store` denotes the current constraint store. This is a very important deduction as it makes the constraint store inconsistent with `Y in 10..14`. So `Nout < 1,Y = X` is posted and the final domains are `N in 1..1, X in 10..12, Y in 10..12`. This example points out that approximating the behaviour of the loop with function w^∞ is crucial to deduce information.

On the examples of sections 2.3 and 2.4 some interesting deductions are done. For the *sum* example, when S is instantiated the value of I is computed. If no value exist, the filtering algorithm fails. Deductions are done even with partial information: `sum(S,I), S in 50..60` leads to $S = 55, I = 10$.

On the request `gcd(X,Y,Z), X in 1..10, Y in 10..20, Z in 1..1000`, the filtering algorithm reduces the bounds of Z to 1..10. Again, this deduction is done thanks to the w^∞ function, which infers the relations $Z \leq X$ and $Z \leq Y$. If we add other constraints, which would be the case in a problem that would use the *gcd* constraint, we obtain more interesting deductions. For example, if we add the constraint $X = 2 * Y$, then the filtering algorithm deduces that Z

is equal to Y. On each of the above examples, the required computation time is not greater than 30 ms.

Another important point is that approximating loops also allows the filtering algorithm to fail instead of non terminating in some cases. Consider this very simple example that infinitely loops if X is lower than 10.

```
loop(X,Xn) :-
   w([X],[Xin],[Xout],[Xn],
        X < 10,
        [Xout = Xin])
```

Suppose that we post the following request, `X < 0, loop(X,Xn)`, and apply the case reasoning. As we can always prove that the loop must be unfolded, the algorithm does not terminate. However, the filtering algorithm can be extended to address this problem. The idea is to compute an approximation of the loop after a given number of iterations instead of unfolding more and more the loop. On the `loop` example, this extension performs well. Indeed the approximation infers `Xn < 0`, which suffices to show that the condition will never be satisfied and thus the filtering algorithm fails. If the approximation cannot be used to prove non-termination, then the algorithm returns the approximation or continue iterating, depending on what is most valuable for the user: having a sound approximation of the loop or iterating hoping that it will stop.

4 Background

This Section gives some background on abstract interpretation. It first presents the general framework. Then, polyhedra abstract domain is presented. Finally, the notion of linear relaxation is detailed.

4.1 Abstract Interpretation

Abstract Interpretation is a framework introduced in [6] for inferring program properties. Intuitively, this technique consists in executing a program with abstract values instead of concrete values. The abstractions used are such that the abstract result is a sound approximation of the concrete result. Abstract interpretation is based upon the following theory.

A lattice $\langle L, \sqsubseteq, \sqcap, \sqcup \rangle$ is complete iff each subset of L has a greatest lower bound and a least upper bound. Every complete lattice has a least element $\bot$ and a greatest element $\top$. An ascending chain $p_1 \sqsubseteq p_2 \sqsubseteq \ldots$ is a potentially infinite sequence of ordered elements of L. A chain eventually stabilizes iff there is an i such that $p_j = p_i$ for all $j \geq i$. A lattice satisfies the ascending chain condition if every infinite ascending chain eventually stabilizes. A function $f : L \to L$ is monotone if $p_1 \sqsubseteq p_2$ implies $f(p_1) \sqsubseteq f(p_2)$. A fixed point of f is an element p such that $f(p) = p$. In a lattice satisfying ascending chain condition, the least fixed point $lfp(f)$ can be computed iteratively: $lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$

The idea of abstract interpretation is to consider program properties at each program point as elements of a lattice. The relations between the program properties at different locations are expressed by functions on the lattice. Finally, computing the program properties consists in finding the least fixed point of a set of functions.

Generally, interesting program properties at a given program point would be expressed as elements of the lattice $\langle \mathcal{P}(\mathbb{N}), \subseteq, \cap, \cup \rangle$ (if variables have their values in $\mathbb{N}$). However, computing on this lattice is not decidable in the general case and the lattice does not satisfy the ascending chain condition. This problem often appears as soon as program properties to be inferred are not trivial. This means that the fixed points must be approximated. There are two ways for approximating fixed points. A static approach consists in constructing a so-called abstract lattice $\langle M, \sqsubseteq_M, \sqcap_M, \sqcup_M \rangle$ with a Galois connection $\langle \alpha, \gamma \rangle$ from L to M. $\alpha : L \to M$ and $\gamma : M \to L$ are respectively an abstraction and concretization function such that $\forall l \in L, l \sqsubseteq \gamma(\alpha(l))$ and $\forall m \in M, m \sqsubseteq_M \alpha(\gamma(m))$. A Galois connection ensures that fixed points in L can be soundly approximated by computing in M. A dynamic approximation consists in designing a so-called widening operator (noted ∇) to extrapolate the limits of chains that do not stabilize.

4.2 Polyhedra abstract domain

One of the most used instanciation of abstract interpretation is the interpretation over the polyhedra abstract domain, introduced in [7]. On this domain, the set of possible values of some variables is abstracted by a set of linear constraints. The solutions of the set of linear constraints define a polyhedron. Each element of the concrete set of values is a point in the polyhedron. In this abstract domain, the join operator of two polyhedra is the convex hull. Indeed, the smallest polyhedron enclosing two polyhedra is the convex hull of these two polyhedra. However, computing the convex hull of two polyhedra defined by a set of linear constraints requires an exponential time in the general case.

Recent work suggest to use a join operator that over-approximates the convex hull [15]. Figure 6 shows two polyhedra with their convex hull and weak join.

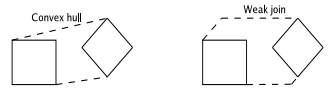


Fig. 6. Convex Hull vs Weak Join

Intuitively, the weak join of two polyhedra is computed in three steps. Enlarge the first polyhedron without changing the slope of the lines until it encloses the second polyhedron. Enlarge the second polyhedron in the same way. Do the intersection of these two new polyhedra.

In many works using abstract interpretation on polyhedra, the standard widening is used. The standard widening operator over polyhedra is computed as follows: if P and Q are two polyhedra such that $P \sqsubseteq Q$. Then, the widening $P \nabla Q$ is obtained by removing from P all constraints that are not entailed in Q. This widening is efficient but not very accurate. More accurate widening operators are given in [1].

4.3 Linear Relaxation of constraints

Using polyhedra abstract interpretation requires us to interpret non linear constraints on the domain of polyhedra. Existing techniques aim at approximating non linear constraints with linear constraints. In our context, the only sources of non linearity are multiplications, strict inequalities and disequalities. These constraints can be linearized as follows:

multiplications Let $\underline{X}$ and $\overline{X}$ be the lower and upper bounds of variable X. A multiplication $Z = X * Y$ can be approximated by the conjunction of inequalities [12]:

$$(X - \underline{X})(Y - \underline{Y}) \geq 0 \wedge (X - \underline{X})(\overline{Y} - Y) \geq 0 \\ \wedge (\overline{X} - X)(Y - \underline{Y}) \geq 0 \wedge (\overline{X} - X)(\overline{Y} - Y) \geq 0$$

This constraint is linear as the product $X * Y$ can be replaced by Z. Fig.7 shows a slice of the relaxation where $Z = 1$. The rectangle corresponds to the bounding box of variables X, Y, the dashed curve represents exactly $X * Y = 1$, while the four solid lines correspond to the four parts of the inequality.

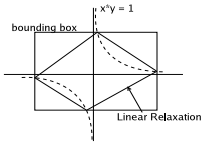


Fig. 7. Relaxation of the multiplication constraint

strict inequalities and disequalities Strict inequalities $X < Var$ (resp. $X > Var$) can be rewritten without approximation into $X \leq Var - 1$ (resp. $X \geq Var + 1$), as variables are integers. Disequalities are considered as disjunctions of inequalities. For example, $X \neq Y$ is rewritten into $X =< Y - 1 \vee X >= Y + 1$. Adding the bounds constraints on X and Y and computing the convex hull of the two disjuncts leads to an interesting set of constraints. For example, if X and Y are both in 0..10, the relaxation of $X \neq Y$ is $X + Y \geq 1 \wedge X + Y \leq 19$.

5 Using abstraction in the filtering algorithm of w

In this section, we detail how abstract interpretation is integrated in the w filtering algorithm. Firstly, we show that solutions of w can be computed with a fixed point computation. Secondly, we explain how abstract interpretation over polyhedra allows us to compute an abstraction of these solutions. Finally, the implementation of the w^∞ function is presented.

5.1 Solutions of w as the result of a fixed point computation

Our problem is to compute the set of solutions of a w constraint:

$$Z = \{((x_1, \ldots, x_n), (x_1^f, \ldots, x_n^f)) \mid \\ w((x_1, \ldots, x_n), In, Out, (x_1^f, \ldots, x_n^f), Cond, Do)\}$$

Let us call S_i the possible values of the loop variables after i iterations in a loop. When $i = 0$ possible variables values are the values that satisfy the domain constraint of $Init$ variables. We call S_{init} this set of values. Thus $S_0 = S_{init}$. Let us call T the following set:

$$T = \{((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \mid (x_1, \ldots, x_n) \in S_{init} \wedge \exists i (x'_1, \ldots, x'_n) \in S_i\}$$

T is a set of pairs of lists of values (l, m) such that initializing variables of the loops with values l and iterating the loop a finite number of times produce the values m. The following relation holds

$$Z = \{(Init, End) \mid (Init, End) \in T \wedge End \in sol(\neg Cond)\}$$

where $sol(C)$ denotes the set of solutions of a constraint C. The previous formula expresses that the solutions of the w constraint are the pairs of lists of values (l, m) such that initializing variables of the loops with values l and iterating the loop a finite number of times leads to some values m that violate the loop condition.

In fact, T is the least fixed point of the following equation:

$$T^{k+1} = T^k \cup \{(Init, Y) \mid (Init, X) \in T^k \wedge (X, Y) \in sol(Cond \wedge Do)\} \quad (1)$$

$$T^0 = \{(Init, Init) \mid Init \in S_{init}\} \quad (2)$$

$Cond$ and Do are supposed to involve only In and Out variables. Thus, composing T^k and $sol(Cond \wedge Do)$ is possible as they both are relations between two lists of variables of length n.

Following the principles of abstract interpretation this fixed point can be computed by iterating Equation 1 starting from the set T^0 of Equation 2.

For the simple constraint: `w([X],[In],[Out],[Y],In < 2,[Out = In+1])` and with the initial domain `X in 0..3`, the fixed point computation proceeds as follows.

$$
\begin{aligned}
T^0 &= \{(0,0),(1,1),(2,2),(3,3)\}\\
T^1 &= \{(0,1),(1,2)\} \cup T^0\\
&= \{(0,0),(0,1),(1,1),(1,2),(2,2),(3,3)\}\\
T^2 &= \{(0,1),(0,2),(1,2)\} \cup T^1\\
&= \{(0,0),(0,1),(0,2),(1,1),(1,2),(2,2),(3,3)\}\\
T^3 &= T^2
\end{aligned}
$$

Consequently, the solutions of the w constraint are given by

$$
\begin{aligned}
Z &= \{(X,Y) \mid (X,Y) \in T^3 \wedge Y \in sol(In \geq 2)\}\\
&= \{(0,2),(1,2),(2,2),(3,3)\}
\end{aligned}
$$

Although easy to do on the example, iterating the fixed point equation is undecidable because Do can contain others w constraints. Thus, Z is not computable in the general case.

5.2 Abstracting the fixed point equations

We compute an approximation of T using the polyhedra abstract domain. Let P be a polyhedron that over-approximates T, which means that all elements of T are points of the polyhedron P. Each list of values in the pairs defining T has a length n thus P involves $2n$ variables. We represent P by the conjunction of linear equations that define the polyhedron.

The fixed point equations become:

$$P^{k+1}(Init, Out) = P^k \sqcup (P^k(Init, In) \wedge Relax(Cond \wedge Do))_{Init,Out} \quad (3)$$

$$P^0(Init, Out) = \alpha(S_{init}) \wedge Init = Out \quad (4)$$

Compared to equations 1 and 2, the computation of the set of solutions of constraint C is replaced by the computation of a relaxation of the constraint C. $Relax$ is a function that computes linear relaxations of a set of constraints using the relaxations presented in Section 4.3. P_{L_1,L_2} denotes the projection of the linear constraints P over the set of variables in L_1 and L_2. Projecting linear constraints on a set of variables S consists in eliminating all variables not belonging to S. Lists equality $L = M$ is a shortcut for $\forall i \in [1,n] L[i] = M[i]$, where n is the length of the lists and $L[i]$ is the ith element of L. $P_1 \sqcup P_2$ denotes the weak join of polyhedron P_1 and P_2 presented in Section 4.2.

In Equation 4, S_{init} is abstracted with the α function. This function computes a relaxation of the whole constraint store and projects the result on $Init$ variables.

An approximation of the set of solutions of a constraint w is given by

$$Q(Init, In) = P(Init, In) \wedge Relax(\neg Cond) \quad (5)$$

We detail the abstract fixed point computation on the same example as in the previous section. As the constraints $Cond$ and Do are almost linear their relaxation is trivial: $Relax(Cond \wedge Do) = X_{in} \leq 1, X_{out} = X_{in} + 1$. X_{in} is only constrained by its domain, thus $\alpha(S_{init}) = X_{in} \geq 0 \wedge X_{in} \leq 3$. The fixed point is computed as follows

$$
\begin{aligned}
P^0(X_{in}, X_{out}) &= X_{in} \geq 0 \wedge X_{in} \leq 3 \wedge X_{in} = X_{out}\\
P^1(X_{in}, X_{out}) &= (P^0(X_{in}, X_0) \wedge X_0 \leq 1 \wedge X_{out} = X_0 + 1)_{X_{in},X_{out}}\\
&\quad \sqcup P^0(X_{in}, X_{out})\\
&= (X_{in} \geq 0 \wedge X_{in} \leq 1 \wedge X_{out} = X_{in} + 1) \sqcup P^0(X_{in}, X_{out})\\
&= X_{in} \geq 0 \wedge X_{in} \leq 3 \wedge X_{out} \leq X_{in} + 1 \wedge X_{out} \geq X_{in}\\
P^2(X_{in}, X_{out}) &= (P^1(X_{in}, X_1) \wedge X_1 \leq 1 \wedge X_{out} = X_1 + 1)_{X_{in},X_{out}}\\
&\quad \sqcup P^1(X_{in}, X_{out})\\
&= (X_{in} \geq 0 \wedge X_{in} \leq 3 \wedge X_{in} \leq X_{out} - 1) \sqcup P^1(X_{in}, X_{out})\\
&= X_{in} \geq 0 \wedge X_{in} \leq 3 \wedge X_{out} \leq X_{in} + 2 \wedge X_{out} \geq X_{in} \wedge X_{out} \leq 4\\
P^3(X_{in}, X_{out}) &= (P_2(X_{in}, X_2) \wedge X_2 \leq 1 \wedge X_{out} = X_2 + 1)_{X_{in},X_{out}}\\
&\quad \sqcup P^2(X_{in}, X_{out})\\
&= (X_{in} \geq 0 \wedge X_{in} \leq 3 \wedge X_{in} \leq X_{out} - 1) \sqcup P^2(X_{in}, X_{out})\\
&= P^2(X_{in}, X_{out})
\end{aligned}
$$

Figure 8 shows the difference between the exact fixed point computed with the exact equations and the approximate fixed point. The points correspond to elements of T^3 whereas the grey zone is the polyhedron defined by P^3.

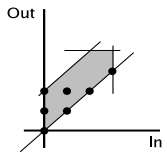


Fig. 8. Exact vs approximated fixed point

An approximation of the solutions of the w constraint is

$$
\begin{aligned}
Q &= P^3(X_{init}, X_{end}) \wedge X_{end} \geq 2\\
&= X_{end} \geq 2 \wedge X_{end} \leq 4 \wedge X_{in} \leq X_{end} \wedge X_{in} \leq 3 \wedge X_{in} \geq X_{end} - 2
\end{aligned}
$$

On the previous example, the fixed point computation converges but it is not always the case. Widening can address this problem. The fixed point equation becomes:

$$
\begin{aligned}
P^{k+1}(Init, Out) = P^k(Init, Out) \nabla\\
(P^k \sqcup (P^k(Init, In) \wedge Relax(Cond, Do))_{Init,Out})
\end{aligned}
$$

In this equation ∇ is the standard widening operator presented in Section 4.2.

5.3 w^∞: implementing the approximation

In Section 3, we have presented the filtering algorithm of the w operator. Here, we detail more concretely the integration of the abstract interpretation over polyhedra into the constraint combinator w via the w^∞ function.

w^∞ is an operator that performs the fixed point computation and communicates the result to the constraint store. Figure 9 describes the algorithm. All the operations on linear constraints are done with the *clpq* library [10].

Input:
$Init, In, Out, End$ vectors of variables
$Cond$ and Do the constraints defining the loop
A constraint store (X, C, B)
Output:
An updated constraint store

w^∞ :
1 $P^{i+1} := project(relax(C, B), [Init]) \wedge Init = Out$
2 **repeat**
3 $\quad P^i := P^{i+1}$
4 $\quad P^j := project(P^i \wedge relax(Cond \wedge Do, B), [Init, Out])$
5 $\quad P^k := weak\_join(P^i, P^j)$
6 $\quad P^{i+1} := widening(P^i, P^k)$
7 **until** $includes(P^i, P^{i+1})$
8 $Y := P^{i+1} \wedge relax(\neg Cond, B)$
9 $(C', B') := concretize(Y)$
10 **return** $(X, C' \wedge w(Init, In, Out, End, Cond, Do), B')$

Fig. 9. The algorithm of w^∞ operator

This algorithm summerizes all the notions previously described. Line 1 computes the initial value of P. It implements the α function introduced in Equation 4. The $relax$ function computes the linear relaxation of a constraint C given the current variables domains, B. When C contains another w combinator, the corresponding w^∞ function is called to compute an approximation of the second w. The $project(C, L)$ function is a call to the Fourier variable elimination algorithm. It eliminates all the variables of C but variables from the list of lists L. Lines 2 to 7 do the fixed point computation following Equation 3. Line 6 performs the standard widening after a given number of iterations in the repeat loop. This number is a parameter of the algorithm. At Line 7, the inclusion of P^{i+1} in P^i is tested. $includes(P^i, P^{i+1})$ is true iff each constraint of P^i is entailed by the constraints P^{i+1}.

At line 8, the approximation of the solution of w is computed following Equation 5. Line 9 concretizes the result in two ways. Firstly, the linear constraints are turned into finite domain constraints. Secondly, domains of End variables are reduced by computing the minimum and maximum values of each variable in the linear constraints Y. These bounds are obtained with the simplex algorithm.

6 Discussion

The polyhedra abstract domain is generally used differently from what we presented. Usually, a polyhedron denotes the set of linear relations that hold between variables at a given program point. As we want to approximate the solutions of a w constraint, our polyhedra describe relations between input and output values of variables and, thus, they involve twice as many variables. In abstract interpretation, the analysis is done only once whereas we do it each time a w operator is awoken. Consequently, we cannot afford to use standard libraries to handle polyhedra, such as [2], because they use the dual representation, which is a source of exponential time computations. Our representation implies, nevertheless, doing many variables elimination with the Fourier elimination algorithm. This remains costly when the number of variables grows. However, the abstraction on polyhedra is only one among others. For example, abstraction on intervals is efficient but leads to less accurate deductions. The octagon abstract domain [13] could be an interesting alternative to polyhedra as it is considered to be a good trade-off between accuracy and efficiency.

Generalized Propagation [14] infers an over-approximation of all the answers of a CLP program. This is done by explicitely computing each answer and joining these answers on an abstract domain. Generalized Propagation may not terminate because of recursion in CLP programs. Indeed, no widening techniques are used. In the same idea, the 3r's are three principles that can be applied to speed up CLP programs execution [11]. One of these 3r's stands for *refinement*, which consists in generating redundant constraints that approximate the set of answers. Refinement uses abstract interpretation, and more specifically widenings, to compute on abstract domains that have infinite increasing chains. Hence, the analysis is guaranteed to terminate. Our approach is an instantiation of this theoretical scheme to the domain of polyhedra.

7 Conclusion

We have presented a constraint combinator, w, that allows users to make a constraint from an imperative loop. We have shown examples where this combinator is used to implement non trivial arithmetic constraints. The filtering algorithm associated to this combinator is based on case reasoning and fixed point computation. Abstract interpretation on polyhedra provides a method for approximating the result of this fixed point computation. The results of the approximation are crucial for pruning variable domains. On many examples, the deductions made by the filtering algorithm are considerable, especially as this algorithm comes for free in terms of development time.

Acknowledgements

We are indebted to Bernard Botella for his significant contributions to the achievements presented in this paper.

References

1. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. of the Static Analysis Symp. (SAS'03)* 337–354, 2003.
2. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *Proc. of the Static Analysis Symp. (SAS'02)*, 213–229. Springer, 2002.
3. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Graph properties based filtering. In *Proc. of the Int. Conf. on Principles and Practice of Constraint Progr. (CP'06)*, 59–74. Springer, 2006.
4. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *Proc. of the Int. Conf. on Principles and Practice of Constraint Progr. (CP'04)*, 107–122. Springer, 2004.
5. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. of the Int. Symp. on Progr. Lang.: Implementations, Logics, and Programs (PLILP'97)*, 191–206. Springer, 1997.
6. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Symp. on Principles of Progr. Lang. (POPL'77)*, 238–252. ACM, 1977.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of Symp. on Principles of Progr. Lang. (POPL'78)*, 84–96. ACM, 1978.
8. T. Fruhwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Progr., Journal of Logic Progr.*, 37(1-3), 1998.
9. A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *First Int. Conf. on Computational Logic (CL'00)*, 399–413, 2000.
10. C. Holzbaur. *OFAI clp(q,r) Manual*. Austrian Research Institute for Artificial Intelligence, Vienna, 1.3.3 edition.
11. K. Marriott and P. J. Stuckey. The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proc. of Symp. on Principles of Progr. Lang. (POPL'93)*, 334–344. ACM, 1993.
12. G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part 1 - convex underestimating problems. *Math. Progr.*, 10:147–175, 1976.
13. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation Journal*, 19:31–100. Springer, 2006.
14. T. Le Provost and M. Wallace. Domain independent propagation. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems. (FGCS'92)*, 1004–1011, 1992.
15. S. Sankaranarayanan, M. A. Colòn, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *Proc. of the Verification, Model Checking, and Abstract Interpretation Conf. (VMCAI'06)*, 115–125. Springer, 2006.
16. J. Schimpf. Logical loops. In *Proc. of the Int. Conf. on Logic Progr. (ICLP'02)*, 224–238. Springer, 2002.
17. C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, 118–132. Springer-Verlag, 2006.

Enfin, nous avons proposé très récemment un résolveur de contraintes dédié aux calculs modulaires sur les entiers machine. Notre travail dans ce domaine a été motivé par la limitation récurrente des résolveurs existants à traiter de manière appropriée les calculs modulaires silencieux de la plupart des langages de programmation utilisés dans les systèmes critiques. Nous proposons dans l'article suivant une théorie du calcul d'intervalles, nommée "Clockwise intervals" adaptée aux calculs modulaires.

A. Gotlieb, M. Leconte, and B. Marre. *Constraint solving on modular integers*. **In Proc. of the 9th Int. Workshop on Constraint Modeling and Reformulation (ModRef'10), co-located with CP'2010, St Andrews, Scotland, Sept. 2010.**

Constraint solving on modular integers

Arnaud Gotlieb[1], Michel Leconte[2], and Bruno Marre[3]

[1] INRIA Rennes Bretagne Atlantique, Campus Beaulieu, 35042 Rennes, France
arnaud.gotlieb@inria.fr
[2] ILOG Lab, IBM France, Gentilly, France
leconte@ibm.fr
[3] CEA, LIST, Gif-sur-Yvette, F-91191, France
marre@cea.fr

Abstract. Constraint solving over finite-sized integers involves the definition of propagators able to capture modular (a.k.a. wrap-around) integer computations. In this paper, we propose efficient propagators for a fragment of modular integer constraints including adders, multipliers and comparators. Our approach is based on the original notion of *Clockwise Interval* for which we define a complete arithmetic. We also present three distinct implementations of modular integer constraint solving in the context of software verification[4].

1 Introduction

Using constraint solving to automatically generate program inputs is an emerging trend in software verification. In the last decade, several tools based on Finite Domains (FD) constraint solving were proposed that perform test inputs generation for C programs (e.g., InKa [7], PathCrawler [9]), test case generation for reactive programs (e.g., GATEL [10]), or property-oriented software verification (e.g., CPBPV [4], Euclide [6]). In these tools, automated verification of intensive integer computations involves solving constraints over finite-sized integers. As an example, consider the problem of reaching[5] statement 2 in the program of Fig.1 that requires solving the decision `buf+len < buf` over unsigned 32-bits integers. A naive translation of the decision of statement 1 as constraint $buf + len < buf$ where buf belongs to $0..2^{32} - 1$ and $len = 2^{31}$, yields an incorrect result saying that statement 2 is unreacheable. In fact, it is trivial to see that this constraint is unsatisfiable when it is interpreted over Finite Domains (FD). However, statement 2 can be reached by selecting a test value such as $buf = 2^{31}$, as $2^{31} + 2^{31} = 2^{32}$ corresponds to value 0 in unsigned 32-bits integer arithmetic. Note also that simplifying $buf + len < buf$ in $len < 0$ is forbidden in this arithmetic. The overall reason is that decision 2 should be rather interpreted as $buf + len < buf \ mod(2^{32})$. In fact, this problem is sometimes reported to as the "wrapping effect" and it turned out that programmers who take care of possible integer overflows routinely write programs that use this effect. In Fig.1, statement 2 can only be reached by a wrapping behaviour. Unfortunately, all the previously mentionned tools that exploits constraint techniques for automated test data generation or property-oriented verification simply ignore this wrapping effect. In fact, as soon as finite domains are specified for each input and intermediate variable, these tools consider that programs with integer overflows are necessarily incorrect and should be rejected. This is obviously abusive and often conducts to report false negatives.

```
unsigned long len = 2147483648;     % Equal to 2^31
void f(unsigned long buf) {
1.        if (buf + len < buf) {
2.                                  ...
```

Fig. 1. Program taking care of integer overflow

This paper adresses this problem by providing efficient constraint solving over modular integer computations. We propose bound-consistency propagators for a linear fragment of these constraints that includes adders, multipliers and comparators. Our approach is based on the original notion of *Clockwise Interval* that captures the wrapping effect by considering intervals with modular integer bounds. An example of such *Clockwise Interval* is the interval $[7, 2]_8$ that represents all the integers x such that $x \ mod \ 8 = 7, x \ mod \ 8 = 0, x \ mod \ 8 = 1, x \ mod \ 8 = 2$. For these Clockwise Intervals, we give a complete arithmetic that has not been published elsewhere.

In the context of the ANR CAVERN project[6] we independently built three distinct implementations of modular integer constraint solving that are variations of clockwise interval arithmetics. These implementations of modular integer constraint solving are used in three software verification tools. Our first implementation called MAXC is used in the context of automatic test input generation for C programs. It implements bound-consistency filtering for a linear fragment of modular integer constraints. For example, for constraint $buf \oplus 2^{31} < buf$ where $\oplus$ denotes modular addition over 32-bit integers, MAXC automatically prunes the domain of buf to the Clockwise Interval $[2^{31}, 2^{32} - 1]_{2^{32}}$, removing half of the variation domain of buf. Our second implementation called JSOLVER [8] is intented to perform automatic analysis of rule-based programs. JSOLVER is based on classic intervals but it takes into account modular integer computations. A comparison with the Clockwise Interval arithmetics shows that JSOLVER is efficient but not optimal when computing local-consistencies over these constraints. Finally, the third implementation is called COLIBRI and it enables automatic test data generation for reactive programs [10]. COLIBRI implements modular integer constraints on domains represented as union of classic intervals.

[4] This work is supported by ANR-07-SESUR-003 CAVERN Project
[5] Reachability is a fundamental problem in software program verification.
[6] cavern.inria.fr

The rest of the paper is organized as follows: Sec.2 introduces the notations and the formal definitions used in the rest of the paper. Sec.3 presents bound-consistency filtering on modular integer constraints. Sec.4 describes our three distinct implementations and discusses their relations with Clockwise Intervals. Finally, Sec.5 concludes and draws several perspectives to this work.

2 Preliminaries

2.1 Notations

Let $\mathbb{Z}$ denote the set of integers and $\mathbb{Z}_b$ denote the finite set of integers modulo b. For any $x \in \mathbb{Z}$ and $y \in \mathbb{Z}^*$, x mod y denotes the integer r such that $\exists q \in \mathbb{Z}$ $r = x - y * q$ and $0 \leq r < y$, while x quo y denotes q the quotient. In the following, we will fix $b = 2^n$ where n is any non-negative integer. Since $\mathbb{Z}_b$ consists of residue classes, several representations are possible. In this paper, we will consider two representations that can be used to emulate integral computations in imperative languages such C or Java: the unsigned representation $\{0, 1, \ldots, b-1\}$ and the signed representation $\{-\frac{b}{2}, \ldots, -1, 0, 1, \ldots, \frac{b}{2} - 1\}$. For the sake of simplicity, we will use the unsigned representation (unless it is mentionned otherwise).

In the context of integer-based manipulations, a classic interval noted $x..y$ where $x, y \in \mathbb{Z}$ and $x \leq y$ denotes the finite ordered set $\{x, x+1, \ldots, y-1, y\}$.

Definition 1 (Width). *The width of an interval $x..y$ is an integer, defined as follows: $wid(x..y) \triangleq y - x$.*

2.2 Clockwise Interval

Definition 2 (Clockwise Interval). *Let x and y be two integers modulo b, a Clockwise Interval (CI) is noted $[x, y]_b$ and denotes the set $\{x, x+1 \ mod\ b, \ldots, y-1 \ mod\ b, y\}$.*

It differs from classic interval in that any of its element is a residue class of integer modulo b. Furthermore, the bound y is not required to be greater than x as the set $\{x, x+1 mod b, \ldots, y-1 mod b, y\}$ is unordered. By convention, we consider that $[0, b-1]_b$ is the canonical representation of $\mathbb{Z}_b$ itself. Note that other representations exist: $[1, 0]_b$, $[2, 1]_b$,...,$[b-1, 0]_b$. Clockwise Intervals that have a positive or null width are called *proper* CIs, while others are called *improper* CIs. The width of a CI is defined by extending the definition of width over classic intervals, by using the canonical representation: $wid([x, y]_b) = wid(x..y)$. Note that width can then becomes negative in this case. The set of clockwise intervals over $\mathbb{Z}_b$ is finite. It is composed of $\{[]_b, [0, 0]_b, \ldots, [b-1, b-1]_b, [0, 1]_b, [1, 0]_b, \ldots, [b-2, b-1]_b, [b-1, b-2]_b, \ldots, [0, b-1]_b\}$, where $[]_b$ denotes the empty clockwise interval.

Definition 3 (Cardinality). *Let $[x, y]_b$ be a CI, then its cardinality is an integer modulo b defined as: $card([x, y]_b) \triangleq (y - x + 1) \ mod\ b$.*

By convention, $card([0, b-1]_b) = b$ and $0 < card([x, y]_b) \leq b$. For example, $card([7, 0]_8) = 2$ while $wid([7, 0]_8) = -7$. The following property immediatly holds:

Proposition 1. *A CI $[x, y]_b$ contains exactly $card([x, y]_b)$ elements, if represented over $[1, b]_b$.*

Proof. If $y \geq x$, then the set $[x, y]_b = \{x, x+1, \ldots, y-1, y\}$ is ordered and contains $y - x + 1$ elements. The special case where $y - x + 1 = b$ corresponds to the CI $[0, b-1]_b$ and then $card([0, b-1]_b) = b$.
If $y < x$, then $[x, y]_b = \{x, x+1, \ldots, b-1\} \cup \{0, 1, \ldots, y\}$ and so, it contains $(b - x) + (y + 1)$ elements. In this case, $b - x + y + 1 \equiv y - x + 1$ mod b that gives the expected result.

2.3 Building clockwise intervals

A classic interval can be converted into a CI by using the following formula:

$$x..y \text{ mod } b \triangleq \begin{cases} [0, b-1]_b & \text{if } wid(x..y) \geq b \\ [x \ mod\ b, y \ mod\ b]_b & \text{otherwise} \end{cases}$$

We define the *hull* of a set of modular integers as being the smallest Clockwise Interval w.r.t. cardinality, that contains all the elements of the set. By convention, proper clockwise intervals are considered smaller than improper ones when they have same cardinality. Formally,

Definition 4 (Hull). *Let $S = \{x_1, \ldots, x_p\}$ be a subset of $\mathbb{Z}_b$, the hull of S is a CI noted $\Box S$, defined as:*

$$\Box S \triangleq Inf_{card}(\{[x_i, x_j]_b | \{x_1, \ldots, x_p\} \subseteq [x_i, x_j]_b)$$

Building an algorithm from this definition yields an untractable procedure as it would require considering $p!$ possible combinations of the bounds. Fortunately, we have the following proposition:

Proposition 2. *Let $S = \{x_0, \ldots, x_{p-1}\}$ be an* **ordered** *subset of $\mathbb{Z}_b$, and let x_{-1} denotes x_{p-1}, then*

$$\Box S = [x_i, x_{i-1}]_b \text{ where } i \in 0..p-1 \text{ such that } card([x_i, x_{i-1}]_b) \text{ is minimized}$$

Therefore, when S is ordered, $\Box S$ can be computed in linear time w.r.t. size of S.

Proof. The case where $[x_i, x_{i-1}]_b$ is proper, i.e., $x_i = x_0$ and $x_{i-1} = x_p$, is trivial. Let suppose that $[x_i, x_{i-1}]_b$ is an improper CI. Firstly, it is clear that $S \subseteq [x_i, x_{i-1}]_b$ as S is ordered ($\forall i \in 1..p-2, x_0 \leq x_{i-1} \leq x_i \leq x_{p-1}$). Secondly, as $card([x_i, x_{i-1}]_b)$ is minimized, it remains to show that there does not exist a CI $[k, l]_b$ where $j \neq i - 1$ that contains S and that is tighter than $[x_i, x_{i-1}]_b$). If $l > x_{i-1}$ then $x_{i-1} \notin [k, l]_b$ and if $k < x_i$ then $x_i \notin [k, l]_b$, meanning that $l \leq x_{i-1}$ and $k \geq x_i$. By this, we get $card([k, l]_b) \geq card([x_i, x_{i-1}]_b)$ which contradicts the hypothesis.

2.4 Clockwise Interval Arithmetic

Having defined Cl, we now turn on the definition of Clockwise Interval Arithmetic that allows us to perform computations over intervals.

Definition 5 (Addition). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the addition operation, noted $\oplus$, is defined as:*

$$[i,j]_b \oplus [k,l]_b \triangleq \begin{cases} [0,b-1]_b & \text{if } card([i,j]_b) = b \text{ or } card([k,l]_b) = b \\ & \text{or } card([i,j]_b) + card([k,l]_b \geq b \\ [(i+k) \bmod b, (j+l) \bmod b]_b & \text{otherwise} \end{cases}$$

Correction property: $\forall x \in [i,j]_b, \forall y \in [k,l]_b, (x+y) \bmod b \in [i,j]_b \oplus [k,l]_b$.

For example, $[2,3]_8 \oplus [3,2]_8 = [0,7]_8$ while $[2,2]_8 \oplus [3,3]_8 = [5,5]_8$.

Definition 6 (Substraction). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the substraction operation, noted $\ominus$, is defined as:*

$$[i,j]_b \ominus [k,l]_b \triangleq \begin{cases} [0,b-1]_b & \text{if } card([i,j]_b) = b \text{ or } card([k,l]_b) = b \\ & \text{or } card([i,j]_b) + card([k,l]_b \geq b \\ [(i-l) \bmod b, (j-k) \bmod b]_b & \text{otherwise} \end{cases}$$

Correction property: $\forall x \in [i,j]_b, \forall y \in [k,l]_b, (x-y) \bmod b \in [i,j]_b \ominus [k,l]_b$.

For example, we have $[0,1]_8 \ominus [0,1]_8 = [7,1]_8$. Note that $[0,b-1]_b$ is absorbing for $\oplus$ and $\ominus$. For those two operations, similarily to the situation in classic Interval Arithmetic, the computations can be performed on the bounds of Clockwise Intervals. This is no longer the case for multiplication and division, as the tightest CI that encloses all the solutions cannot be computed by using only bounds of its operands in those cases. Let us first define precisely the considered operations:

Definition 7 (Multiplication by a constant k). *Let k be a constant modulo b and $[i,j]_b$ a CI, then the multiplication by k is defined as follows:*

$$k * [i,j]_b \triangleq \square(\{k*i \bmod b, k*(i+1) \bmod b, ..., k*j \bmod b\})$$

Definition 8 (Multiplication). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the multiplication operation, noted $\otimes$, is defined as:*

$$[i,j]_b \otimes [k,l]_b \triangleq \square(\{i*k \bmod b, i*(k+1) \bmod b, ..., (i+1)*k \bmod b, ..., j*l \bmod b\})$$

Definition 9 (Division). *Let $[i,j]_b$ and $[k,l]_b$ be two CI, then the division operation, noted $\oslash$, is defined as:*

$$[i,j]_b \oslash [k,l]_b \triangleq \square(\{i/k \bmod b, i/(k+1) \bmod b, ..., (i+1)/k \bmod b, ..., j/l \bmod b\})$$

As an example, consider the *multiplication by a constant* operation $4 \otimes [2,4]_8$. With the formula, we get $\square(\{4*2 \bmod 8, 4*3 \bmod 8, 4*4 \bmod 8\}) = \square(\{0,4\}) = [0,4]_8$. Unfortunately, the bounds of the resulting CI $[0,4]_8$ cannot be computed by using only the bounds of CI operands as $4*2 \equiv 4*4 \equiv 0 \bmod 8$. Computing the resulting CI by enumerating all the elements of its operands seems unreasonnable in the context of large-sized machine integers. The following subsection describes a method that permits to compute the resulting optimal CI in the case of multiplication by a constant k, without requiring a ful enumeration of the domain of possible values.

2.5 An efficient method for computing optimal CI in the presence of multiplication operators

The method is based on the following notes:

- the structure of $\mathbb{Z}_{2^n}$ is well known: the divisors of 0 are powers of 2 ;
- thanks to proposition 2, $\square(\{x_1, ..., x_p\})$ can be computed efficiently when the set $\{x_1, ..., x_p\}$ is ordered.

Let k be a constant modulo $b = 2^n$, let $[i,j]_b$ be a CI, we describe a method that allows to compute the minimum and the maximum values of $k * [i,j]_b = \square(\{k*i \bmod b, k*(i+1) \bmod b, ..., k*j \bmod b\})$.

We start by eliminating some trivial cases: If $k = 0$, then $k*[i,j]_b = \square(\{0\}) = [0,0]_b$. If $k = 1$, then $k*[i,j]_b = [i,j]_b$. If $i \leq j$ and $k*j < b$, then $k*[i,j]_b = [k*i, k*j]_b$. Let now suppose that k is a constant greater or equal to 2 and $k*j \geq b$ or $i > j$. We have the following proposition:

Proposition 3. *Let $k \neq 2^w$, $q_1 = k*i$ quo b and $q_2 = k*j$ quo b, then:*
$Max(k*[i,j]_b) = b - d$ *where* $d = Min_{q_1 < q \leq q_2}(q*b \bmod k)$ *and*
$Min(k*[i,j]_b) = d'$ *where* $d' = Min_{q_1 < q \leq q_2}(-q*b \bmod k)$.

Proof. (sketch of, partial) Let p be the element of $[i,j]_b$ for which $k*p \bmod b$ is maximized in $\mathbb{Z}_b$, and let q be the smallest value such that $k*p < q*b$, then we consider $d = q*b - k*p$. We claim that $d = q*b \bmod k$ as $0 < p < k$. It remains to find the value of q that minimizes $q*b \bmod k$. As $p \in [i,j]_b$, we know that $q_1 < q \leq q_2$ by definition of q. Therefore we can explore the possible values of q from $q_1 + 1$ to q_2, up to $k-1$ values.

For example, consider $k*[i,j]_b$ where $k = 5$ and $[i,j]_b = [2,7]_8$. Applying Prop.3, we get $q1 = 5*2 \text{ div } 8 = 1$ and $q2 = 5*7 \text{ div } 8 = 4$. For $q = 2,3,4$, computing $r_q = q*b \bmod k$ and $r_{-q} = -q*b \bmod k$ leads to:
$r_2 = 16 \bmod 5 = 1$ and $r_{-2} = -16 \bmod 5 = 4$,
$r_3 = 24 \bmod 5 = 4$ and $r_{-3} = -24 \bmod 5 = 1$,
$r_4 = 32 \bmod 5 = 2$ and $r_{-4} = -32 \bmod 5 = 3$.
The minimum over the r_i is obtained when $q = 2$ and then $Max(5*[2,7]_8) = 8 - r_2 = 1$. For the r_{-i}, it is obtained when $q = 3$ leading to $Min(5*[2,7]_8) = r_{-3} = 1$. Hence, $5*[2,7]_8 = [1,7]_8$ has been computed by exploring only the

divisors of b in $k * i..k * j$, instead of looking at all the double products $k * l$ within the same range.

Finding similar propositions for generalized multiplication and division may be possible, but one can also use Prop.3 to compute over-approximations of the resulting CIs. It suffices to use the bounds of each operand interval as a constant, to apply Prop.3 on each of the four double products, and keep the smallest intersection of results. But note that, optimality is usually lost with this approach.

3 Constraint propagation over Clockwise Intervals

In this section, we define projection functions that allow to perform constraint propagation over CI. As usual in Finite Domains constraint solving, each variable X is associated a finite domain $dom(X)$ of possible values. We consider here that domain are (over-)approximated by CI: $CI(X) \triangleq \Box(dom(X))$.

3.1 Set-based operations over CI

Inclusion, union and intersection of Clockwise Intervals are defined by using their set-theoretic definition counterpart. For example, *inclusion* over CI is defined as follows:

$$[i,j]_b \subseteq [k,l]_b \iff \{i, i+1, \ldots, j\} \subseteq \{k, k+1, \ldots, l\}$$

Note however that union and more surprisingly intersection are not closed over CI. For example, $[5,2]_8 \cap [1,6]_8 = \{1,2,5,6\}$. Hence, we define the *meet operation* as taking the smallest CI that contains all the elements of the intersection:

$$[i,j]_b \bigwedge [k,l]_b \triangleq \Box(\{i, i+1, \ldots, j\} \cap \{k, k+1, \ldots, l\})$$

For example, we got: $[5,2]_8 \bigwedge [1,6]_8 = [1,6]_8$ and $[5,1]_8 \bigwedge [0,6]_8 = [5,1]_8$. The main question is whether these operations can be computed efficiently. The following property helps answering this question:
Let x be an integer modulo b, then $x \in [i,j]_b$ is true iff $x \geq i \wedge x \leq j$ when $[i,j]_b$ is proper and $x \geq i \vee x \leq j$ when $[i,j]_b$ is improper. This property comes directly from definition of CI.

The meet operator $\bigwedge$ As the computations of *meet* is at the core of constraint propagation engine, finding an efficent algorithm is of great importance. The definition given above requires to explore each element of both domains at least once. This can be costly when large domains are involved during constraint propagation. The following proposition offers ways to compute the meet operation more efficently:

Proposition 4. *Let $X = [i,j]_b$ and $Y = [k,l]_b$ be two CI, then $X \bigwedge Y$ is defined as:*

if $wid(X) * wid(Y) = 0$ (suppose for example that $X = [i,i]_b$)

$$X \bigwedge Y = \begin{cases} [i,i] & \text{if } X = [i,i]_b \wedge i \in Y \\ [k,k] & \text{if } Y = [k,k]_b \wedge k \in X \\ []_b & \text{otherwise} \end{cases}$$

if $wid(X) * wid(Y) > 0$ then

$$X \bigwedge Y = \begin{cases} []_b & \text{if } wid(X) > 0 \wedge wid(Y) > 0 \wedge max\{i,k\} > min\{j,l\} \\ [max\{i,k\}, min\{j,l\}]_b & \text{otherwise} \end{cases}$$

if $wid(X) * wid(Y) < 0$ then

$$X \bigwedge Y = \begin{cases} []_b & \text{if } j < k \wedge l < i \\ [k,j]_b & \text{if } j \geq k \wedge l < i \\ [i,l]_b & \text{if } j < k \wedge l \geq i \\ Y & \text{if } j \geq k \wedge l \geq i \\ & \wedge\ card(Y) \leq card(X) \\ X & \text{if } j \geq k, l \geq i \\ & \wedge\ card(X) < card(Y) \end{cases}$$

In these cases, proving that $CI(X) \bigwedge CI(Y) = \Box(\{i, i+1, \ldots, j\} \cap \{k, k+1, \ldots, l\})$ is not difficult.

Note that the situation differs from classic Interval Arithmetic where the intersection of two intervals is always an interval enclosed within its two operands. Here, $[i,j]_b \bigwedge [k,l]_b$ is sometimes not included in both $[i,j]_b$ or $[k,l]_b$. This could be problematic w.r.t. the monotony of projection functions. Fortunately, the meet operation requires to minimize the cardinality of the resulting clockwise interval. Hence, each time a projection function is called on variable X, the cardinality of $CI(X)$ decrases. This ensures the computations progress towards a fixpoint.

$\bigvee$: the join operator The join operation is defined accordingly:

$$[i,j]_b \bigvee [k,l]_b \triangleq \Box(\{i, i+1, \ldots, j\} \cup \{k, k+1, \ldots, l\})$$

Proposition 5. *Let $CI(X) = [i,j]_b$ and $CI(Y) = [k,l]_b$, then $CI(X) \bigvee CI(Y)$ can be defined as follows:*
if $wid(CI(X)) \geq 0 \wedge wid(CI(Y)) \geq 0$ then

$$CI(X) \bigvee CI(Y) = \begin{cases} [i,l]_b & \text{if } card([i,l]_b) \leq card([k,j]_b) \\ [k,j]_b & \text{otherwise} \end{cases}$$

Note that these two operations ($\bigwedge$,$\bigvee$) give the CI set a structure of a finite lattice.

3.2 Relations over CI

Let X, Y be two variables over $\mathbb{Z}_b$, the relation $X = Y$ leads to prune $CI(X)$ and $CI(Y)$ with the following rule: $CI(X), CI(Y) \leftarrow CI(X) \bigwedge CI(Y)$. In CI Arithmetic, the relation $X \leq Y$ leads to prune $CI(X) = [i, j]_b$ and $CI(Y) = [k, l]_b$ with the rule $CI(X) \leftarrow CI(X) \bigwedge [0, max(CI(Y))]$. Other relations can easily be derived from these ones.

3.3 Bound-consistency for modular integer constraints

From the formula given above, one can derive practical algorithms to perform bound-consistency on modular integer constraints. The simplest approach is to implement propagators on Clockwise Intervals within an AC-3 propagation algorithm. Once a CI becomes empty, then the constraint system is shown as being unsatisfiable. If none CI become void, then the resulting CIs encompass all the solutions of modular integer constraints.

For the linear fragment of modular integer constraints (i.e., addition, substraction, multiplication by a constant) this approach maintains optimal CIs at the cost of bounds computations. However, as soon as variable multiplication is encountered, optimality requires time-quadratic exploration of CIs. This is prohibitive in the context of 32-bits or 64-bits integer arithmetic. This problem is similar to the situation in bit-vector arithmetic [2] where variable multiplication requires time-quadratic computations on the number of bits. For these non-linear constraints, as said previously, one can gave up optimality by computing Clockwise Intervals that over-approximate optimal clockwise intervals. In the implementations described below, several propositions are made in this direction.

4 Implementations

In the context of the ANR CAVERN project, three distinct implementations of modular integer constraint solving were done. During this work, it appears that Clockwise Interval may be a unifying notion capturing the essence of modular integer interval computations.

4.1 MAXC

At INRIA Rennes, the Clockwise Interval Arithmetic shown above was directly implemented in MAXC, a solver dedicated to modular constraint solving. In a near future, this solver should be integrated within EUCLIDE [6], an automatic test data generator for critical C programs. The constraint system that is derived from EUCLIDE includes modular constraints based on arithmetic operators (+,-,*,div,mod) and high-level operators such as reification and global constraints dedicated to program verification. We do not detail these operators here as our paper is focussed on modular constraint solving. Propagators in MAXC are implemented in C for efficiency reasons while the general propagation queue is implemented in Prolog. Each variable is associated to a CI and contracting propagators aim at pruning CIs of their inconsistent values. The size of variables that can be represented in MAXC ranges from 1 bit to 64 bits as these are the sizes typically found in primitive types in C. The data structure for encoding CIs maintains cardinality and width:

```
typedef struct {
  USH       empty     ;  /* is an empty domain ?                        */
  USH       sign      ;  /* is a signed domain ?                        */
  USH       size      ;  /* allowed size = 1,2,3,4,8,16,32 or 64 bits   */
  UL        min       ;  /* min_value of domain                         */
  UL        max       ;  /* max_value of domain                         */
  UL        wid       ;  /* absolute value of width of domain           */
  SSH       sign_wid  ;  /* sign of width: SINGLE is 0 (eg [3,3]),
                            PROPER +1 (eg [3,6]),IMPROPER -1 (eg [6,3]) */
  UL        card      ;  /* cardinality of domain. 0 is the whole domain*/
  ULL       basis     ;  /* basis of modular calculus. 0 denotes 2^64   */
} TYPE_LFD ;
```

In this data structure, USH stands for *unsigned short integer* which corresponds to 16-bits integers while UL stands for *unsigned long*, i.e. 32-bits integers. Other keywords can easily be understood as variations of these two. Note that encoding 64-bits integer Clockwise Interval arithmetics is still possible but greater formats cannot be encoded. The solver applies bound-consistency propagators on this data structure for $\oplus, \ominus, \otimes, \ldots$. It maintains optimal CIs for the linear fragment of these constraints. The input format of constraints is an intermediate one, where complex constraints have already been decomposed in simpler ones. Typical requests are of the form:

```
test1 :-                                    % In 3-bits integer arith.,
   solveur:init_env(E),          %  X = 5, Y in 2..7, Z in 5..0, Z = X*Y
   lfd:news([X,Y,Z],int(8),['X','Y','Z'],E),            % should produce
   lfd:equal(const('5'),X),                       % Y in 3..6, Z in 6..7
   lfd:equal(in('2','7'),Y),
   lfd:equal(in('5','0'),Z),
   lfd:equal('*',X,Y,Z),
   solveur:solve(E),
   lfd:affiche([X,Y,Z]).
```

Many operators still have to be implemented in order to capture modular integer constraints coming from C programs, including bit-to-bit operators (e.g., `&, |, ~`) , logical operators (e.g., `&&, ||`), nonlinear operators coming from destrucive assignment (i.e., `i *= i++` that correspond to constraint $i_2 = (i_1 + 1)^2$), and so on.

4.2 JSOLVER

JSolver is a IBM-ILOG Constraint-Based Programming library in (pure) Java. It is derived from the C++ library IBM-ILOG Solver and has been tailored for

the static and dynamic analyses of rule-based programs [3, 8]. Currently, these analyses are performed using an idealized integer arithmetic where modular computations are ignored. Consequently overflows on integers are reported as errors and the corresponding rule-based programs are rejected which is the expected behaviour, as these programs are exploited by end-users and not by developers.

We recently investigated the use of CP to perform static analysis of rules in order to optimize their compilation in a discrimination network [5]. Unlike the above usage, this requires using the program execution semantics where integer overflows are silently done (such as in Java). We report here on our first implementation of bound consistency for integer modular constraints by using classic intervals as defined in mathbooks: a classic interval $a..b$ with integer bounds a and b is the set of integers $\{x|a \leq x \leq b\}$. Let us consider two positive 32-bits integers and suppose we want to determine the range of the sum of these (signed) integers ranging from 1 to $2^{31} - 1$. By using an idealized semantics for integer computations, we get that the sum is ranging from 2 to $2(2^{31} - 1)$. Of course, this range could be exactly represented by using unbounded values such as bigInteger in Java or approximated by $2.. + infinity$. But, taking into account modular integer arithmetic, we found that the sum is actually ranging on $-2^{31}..-2$ union $2..2^{31}-1$. The classic interval which covers all these values is the set of all representable signed values on 32 bits $MIN\_INT..MAX\_INT$. Note that such classic intervals usually over-approximate the results that could be computed using Clockwise Intervals as, for example, the CI $[2, -2]_{2^{32}}$ on signed 32-bits integers corresponds precisely to $-2^{31}..-2$ union $2..2^{31}-1$ that is over-approximated by the classic interval $-2^{31}..2^{31}-1$. To give a flavor of inferences which could be made on classic interval for modular integer arithmetic, let us continue our example by constraining the sum to be greater than -2. Let x, y and z be three signed 32-bits integers such that z, the sum of x and y, is greater than -2. x and y are ranging on $1..MAX\_INT$ and z is ranging on $-1..MAX\_INT$. As the transformation $x = z - y$ (resp. $y = z - x$) is correct in modular integer arithmetic, we actually found that x (resp. y) is ranging on $1..MAX\_INT - 1$. As $z = x + y$, we deduce that z is ranging on $2..MAX\_INT$, then discovering that z is positive.

To formally define what our computations are, let us assume that we are dealing with modular integer with a machine representation ranging from a smaller integer denoted by m and a larger integer here denoted by M. let $x..y$ be a classic interval. u, v represent x and y in a (m, M) computer integer arithmetics if $m \leq u \leq M$, $m \leq v \leq M$ and $x = u + k_u(M - m + 1)$, $y = k_v(M - m + 1)$ for two integers k_u and k_v. We may then introduce a $cast_{m,M}$ function from classic intervals to (m, M) intervals with the following definition:

$$cast_{m,M}(x..y) = \begin{cases} u..v & \text{if } k_u = k_v \\ m..M & \text{if } k_u! = k_v \end{cases}$$

This cast function provides concise definition for modular arithmetic. For example the (best) forward operator for the sum z of x and y is z in $cast_{m,M}(x_{min} + y_{min}..x_{max} + y_{max})$. As $z = x + y$ is equivalent to $x = y - z$ in modular arithmetic, the (best) backward operator is defined by $x\ in\ cast_{m,M}(z_{min} - y_{max}..z_{max} - y_{min})$ and $y\ in\ cast_{m,M}(z_{min} - x_{max}..z_{max} - x_{min})$.

The multiplication by a scalar is not so straightforward. On 32-bits signed integer, the powers of 2 are divisors of 0 and the congruence domains [8] are not preserved. For example, $2 * MIN\_INT = 0$ and $3 * MIN\_INT$ which is equal to MIN_INT is not even divisible by 3. However, solving $ax = b$ on 32-bits signed integers is not so difficult. First we note that if a power of 2 divides a, it should also divide b for the equation to have a solution. By simplifying by this power of 2, say 2^p, we obtain $a'x = b'\ mod\ 2^{(32-p)}$. We find then the inverse u of $a'\ mod\ 2^{(32-p)}$. We infer that $x = (ua')x = u(a'x) = ub'\ mod\ 2^{(32-p)}$. Finally, we obtained a range for x in 32-signed integers and a congruence domain to be propagated. We can apply this method to the solving of ax in $m..M$ on 32-bits integers in a similar way. First, if 2^p divides a, we keep only the multiples of 2^p from $m..Max$. Then we simplify m and M to solve ax in $m'..M'\ mod\ 2^{(32-p)}$ by finding an inverse u of a', leading to x in $u*m'..u*M'\ mod\ 2^{(32-p)}$. Here again, we infer a range for $x\ mod\ 2^{32}$ and a linear congruence to be propagated. To end this short report on our preliminary implementation, we should say that the general multiplication of two variables is propagated by using the cast function. This leads very often to the top approximation of the full integers, being not complete but at least correct. For future implementations, we are thinking of making use of the k_u integer indicators in the cast function definition, as proposed in the tool COLIBRI described below. We also think to switch our implementation from classic intervals to clockwise intervals as the performance should be similar whilst the precision is improved.

4.3 COLIBRI

COLIBRI is a constraint library developped at CEA LIST for its test generation tools: GATeL for the functional testing of LUSTRE/SCADE models [10], PathCrawler for the structural testing of C code [11] and Osmose for the structural testing of binary code [1]. This library provides domains and constraints for integer, real and floating point interval arithmetics. Furthermore, a congruence domain is combined with the integer domain as described in [8].

The integer domain is implemented by union of intervals with finite bounds. These bounds can be any integer since we use big integers provided by the Gnu Multi-Precision library. This representation of integer domains allows to precisely represent improper clockwize intervals. For example, using clockwise intervals we have $[2, 4]_8 \oplus [4, 7]_8 = [6, 3]_8$. This interval corresponds to the following union of classic intervals $0..3 \cup 6..7$ which is denoted by $[0..3, 6..7]$ in COLIBRI.

In order to handle the signed and unsigned integer arithmetics used by computer languages, COLIBRI provides signed and unsigned modular arithmetics operations when the modulo is a positive power of two (i.e., when $b = 2^n$ with $n > 0$). For each operation op in $+, -, \times, div, rem, power$ we provide the operations $op_{s,n}$ and $op_{u,n}$ which correspond to the modular signed and unsigned versions of op. The implementation of these operations uses the following definition of modular operations.

$$\forall (A, B, C) \in {\mathbb{Z}_{2^n}}^3, A\ op_{su,n}\ B = C \equiv (\exists K, AopB = C + K \times 2^n)$$

The range of K can be easily characterized for each $op_{su,n}$ according to 2^n. For example, for the $+_{u,n}$ operation $-1 \leq K \leq 1$, while for the $\times_{u,n}$ operation $0 \leq K \leq 2^n - 1$.

Thus, according to the previous equivalence, the constraint propagators of any modular operation can be implemented with those of non modular operations. This is exactly the way modular operations are implemented in COLIBRI. For example, the constraint $A +_{u,n} B = C$ where variables A, B and C belong to $[0, 2^n - 1]$ is handled by the following conjunction of constraints: $A+B = X \wedge K \times 2^n = Y \wedge X+Y = C$ where the initial domain of K is $[0, 2^n-1]$, Notice that the congruence domain knows that variable Y is a factor of 2^n and as soon as C (resp. Y) handles a congruence one can infer a congruence for Y (resp. C).

For each modular operation, the variable K is a precise indication of underfow/overflow: when $K < 0$ this means that there is an underflow, when $K > 0$ this means that there is an overflow while when $K = 0$ there is no underfow/overflow. Such an indicator could be very helpfull for verification tools when checking computation w.r.t. underfow/overflow. This is why COLIBRI modular constraints handle a supplementary argument UO which abstracts the sign of K: UO belongs to $[-1, 1]$ and has the same sign as K. Any assignment of this variable UO can be used to force underflow ($UO = -1$), overflow ($UO = 1$) of normal computation ($UO = 0$). Moreover, one can force non normal behaviour by stating that $UO <> 0$.

To conclude this short presentation of modular operations in COLIBRI, let us remark that the accuracy of this implementation relies on the use of union of intervals with big integer bounds. This could be considered very expensive for constraints systems involving heavy computations. However, as shown by a recent experiment [2] using SMT-LIB benchmarks our implementation of modular arithmetics is competitive with powerfull SMT solvers.

5 Conclusions and perspectives

In this paper, we introduced Clockwise Intervals as a way to capture modular interger interval computations. We described three distinct implementations of modular integer constraint solving that have applications in program testing and analysis. We have also seen that finding optimal bounds in bound-consistency filtering of modular integer computations is not trivial and often requires approximations. For general multiplication and division, efficient ways to compute optimal bounds still need to be found. On the foundations of the approach, Clockwise Interval appears as a good tool to describe bound-consistency on modular integer computations but its relations with other Interval Arithmetics still need to be studied. On the applications of modular integer constraint solving, experimental evaluation is required in the diverse contexts presented earlier in the paper, namely, automatic test inputs generation for C programs, test case generation for reactive programs and rule-based program analysis. Another perspective of this work concerns the way other dedicated constraint solver could be married with Clockwise Intervals. For example, the recent work of Bardin et al. in [2] showed that dedicated bitvectors operators could be efficiently coupled with classic intervals. It remains to find ways to integrate such arithmetics within Clockwise Intervals.

Acknowledgments

We would like to thank the members of the ANR CAVERN project who participated to our initial discussions on this topic, namely Bruno Berstel, Bernard Botella, Claude Michel, Michel Rueher, and Nicky Williams.

References

1. S. Bardin and P. Herrmann. Structural testing of executables. In *1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08)*, pages 22–31, 2008.
2. S. Bardin, P. Herrmann, and F. Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis (TACAS'10)*, pages 84–98, 2010.
3. B. Berstel and M. Leconte. Using constraints to verify properties of programs. In *2nd Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'10*, 2010. Co-located with ICST'10 in Paris, April.
4. H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.
5. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
6. A. Gotlieb. Euclide: A constraint-based testing platform for critical c programs. In *2th IEEE International Conference on Software Testing, Validation and Verification (ICST'09)*, Denver, CO, Apr. 2009.
7. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proceedings of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.
8. M. Leconte and B. Berstel. Extending a cp solver with congruences as domains for software verification. In *1st Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'06*, 2006. Co-located with CP'06 in Nantes, September.
9. B. Marre, P. Mouy, and N. Williams. On-the-fly generation of k-path tests for c functions. In *Proceedings of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04)*, Linz, Austria, September 2004.
10. Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111:93 – 111, 2005.
11. N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. Dependable Computing - EDCC'05*, 2005.

Les travaux décrits dans ce chapitre ont été menés en grande partie dans le cadre du projet SESUR CAVERN[2] (*Constraints and Abstractions for program VERificatioN*, 2008-2011). Ce projet coordonné par l'INRIA Rennes, a rassemblé le laboratoire I3S de l'Université de Nice Sophia-Antipolis, le CEA Saclay et la société IBM-ILOG Labs, dans le but d'étudier l'intérêt de techniques de calculs sur les domaines abstraits pour la vérification de programmes à base de contraintes.

Ce chapitre clos la partie fondements de notre manuscrit en ayant présenté, au travers de quelques articles, les principaux éléments qui constitue les fondations du *test à base de contraintes*. Bien sûr, ces éléments ont été présentés au travers du prisme de nos contributions, forcément réducteur, mais ils constituent néanmoins selon nous un bon point de départ pour explorer ce domaine de recherche.

[2] `cavern.inria.fr`

Part II

Développements

Chapter 4

Oracles

Contexte

Le test logiciel repose sur la disponibilité d'oracles, c'est à dire de procédures manuelles ou automatisées, permettant de contrôler les sorties attendues d'un programme sous test. Idéalement, chaque oracle devrait être correct et complet, signifiant qu'il devrait être capable de prédire, sans erreur, la sortie attendue quelque soit l'entrée soumise au programme. Cependant, beaucoup de situations réelles montrent que la disponibilité d'un tel oracle, correct et complet, est extrêmement rare. Ainsi que l'a noté Elaine Weyuker dans un article fameux [Weyuker 82], certains programmes sont même considérés comme étant non–testables car il n'est pas possible de leurs trouver un oracle, même incomplet. A titre d'exemple, on peut citer les programmes écrits pour résoudre un problème dont la solution n'est pas connue à l'avance (comme les programmes à contraintes par exemple), certains programmes numériques qui sont instables vis à vis des erreurs d'arrondis dus aux calculs flottants, ou encore les programmes qui calculent le résultat de fonctions mathématiques complexes, incalculable à la main. De plus, certains programmes doivent être testés de manière approfondie, bien que leur code source ou qu'une spécification formelle de leur comportement attendu, ne soit disponible. C'est le cas particulier des librairies et des composants logiciels externes qui sont souvent délivrés sous forme d'exécutables et pour lesquels seule une documentation informelle existe.

Nous nous sommes intéressés à ce problème au travers de la définition d'oracles partiels, c'est à dire de conditions nécessaires mais non toujours suffisantes pour établir la correction du programme sous test. Cette notion d'oracles partiels a connu de nombreux développements dans le cadre du *test métamorphique* [Chan 98, Chen 01, Tse 07]. Les oracles partiels que nous avons considérés sont fondés sur l'utilisation de propriétés de symétrie des programmes. Ces propriétés de symétrie sont des relations de permutation d'entrée-sorties définies par l'utilisateur qui conduisent à partitionner l'espace d'entrée en classes d'équivalence. L'équivalence de deux exécutions à l'intérieur d'une classe peut alors être utilisée comme un oracle partiel. Dans ce chapitre, nous introduisons un paradigme de test logiciel

nommé *test symétrique* qui est utilisé pour vérifier les programmes au travers de ces propriétés de symétrie. Etant donné l'interface d'un programme et une relation de symétrie que le programme est sensé satisfaire, le test symétrique associe une génération automatique de de cas de test avec le contrôle d'oracles partiels pour découvrir des fautes de symétrie dans les programmes. Cette vérification peut-être complètement automatisée comme nous l'avons montrée dans [Gotlieb 03b]. Le *test symétrique* s'appuie sur des résultats classiques de la *théorie des groupes* afin de minimiser le nombre de sorties à contrôler pour vérifier une propriété de symétrie. Cette théorie est une perle mathématique qui est l'outil indispensable pour parler de symétrie. Le *test symétrique* a été appliqué au test de bibliothèque Java et de programmes Java Card, comme détaillé au chapitre 8 de ce mémoire.

A. Gotlieb. *Exploiting symmetries to test programs*. **In IEEE International Symposium on Software Reliability and Enginering (ISSRE'03), Denver, CO, USA, November 2003.**

Exploiting Symmetries to Test Programs

Arnaud Gotlieb
INRIA/IRISA
Campus Beaulieu
35042 Rennes Cedex, FRANCE
Arnaud.Gotlieb@irisa.fr

April 4, 2003

Abstract

Symmetries often appear as properties of many artifical settings. In Program Testing, they can be viewed as properties of programs and can be used to check the correctness of the computed outcomes. In this paper, we consider symmetries to be permutation relations between program executions and use them to automate the testing process. We introduce a software testing paradigm called Symmetric Testing, where automatic test data generation is coupled with symmetries checking to uncover faults inside the programs. A practical procedure for checking that a program satisfies a given symmetry relation is described. The paradigm makes use of Group theoretic results as a formal basis to minimize the number of program executions required by the method. This approach appears to be of particular interest for programs for which neither an oracle, nor any formal specification is available. We implemented Symmetric Testing by using the primitive operations of the Java unit testing tool *Roast* [1]. The experimental results we got on faulty versions of classical programs of the Software Testing community show the effectiveness of the approach.

1 Introduction

Testing imperative programs at the unit level requires to select test data from the input domain, to execute the program with the selected test data and finally to check the correctness of the computed outcomes. For almost three decades, propositions have been made to automate this process. Structural test data generation relies on program analysis to find automatically a test set that guarantee the coverage of some cirteria based on flow graphs [2, 3, 4, 5]. Functional testing is based on the specifications analysis to generate automatically test data [6, 7]. These techniques both require a formal description to be given as input : the source code of programs in the case of structural testing ; the formal specification of programs in the case of functional testing. However there are programs to be tested for which no one of these formal descriptions is available. For example, commercial off-the-shelf components are usually delivered as "black-boxes", i.e. executable objects whose licenses forbid de-compilation back to the source code [8], and informal specification is used most of the time to describe their expected behaviour. In these situations, techniques such as random testing [9], boundary-value analysis [10] or local exhaustive testing [11] can be employed. Random testing aims at selecting randomly the values inside the input domain by using pseudo-random values generators, whereas boundary-value analysis relies on selecting the boundaries of each individual or dependent domains [1] of the input space. Local exhaustive testing requires to identify critical points around which input values will be exhaustively selected. All these methods have in common to focus on the generation

of input values and are based on an underlying assumption which concerns the availability of a correct and complete oracle, i.e. a procedure able to predict the right outcome for any input data. Unfortunately, there are situations where this assumption seems to be unreasonable. As pointed out by Weyuker [12], some programs are considered to be non-testable. These are programs for which it is theoretically possible, but practically too difficult to determine the correct outcome. Consider programs intented to compute a function which is not accurately known or programs for which correct answers are too difficult to compute by hand. Third-party librairies and commercial components fall usually into the former case [13], whereas complex numerical programs fall into the latter [14].

In this paper, we introduce a software testing paradigm, called Symmetric Testing (ST), which aims at testing imperative programs for which neither an oracle, nor any formal description is required. We consider symmetries to be permutation relations between program executions and use them to automate the testing process. Given the interface of a program and a symmetry relation, ST combines automatic test data generation and symmetries checking to uncover faults within the program. Group theoretic results are used as a formal basis, conforming so the well-known adage *"Numbers measure size, Groups measure symmetry"* [15]. As a trivial example, consider the program p intended to compute the greatest common divisor (gcd) of two non-negative integers u and v and suppose that p is tested with the following test datum ($u = 1309, v = 693$), automatically generated by a random test data generator. Although, we all know how to compute the gcd of two integers[1], it is not so easy to predict the expected value of $gcd(1309, 693)$ without the help of a calculator. Fortunately, gcd satisfies a simple symmetry relation :$\forall u \forall v, gcd(u, v) = gcd(v, u)$. So, if $gcd(1309, 693) \neq gcd(693, 1309)$ then the testing process will succeed to uncover a fault without the help of any oracle of gcd. We generalized this idea to obtain a formal definition of symmetry relation on imperative programs. Formally speaking, let p be a program which takes a vector of at least k values as input and returns a vector of at least l values, and let x and y be two vectors then a symmetry relation for p holds if[2] :

$$\forall \theta \in S_k, \exists \eta \in S_l \text{ such as } y = \theta.x \implies p(y) = \eta.p(x)$$

where S_k (resp. S_l) is the symmetric group acting on k elements of x (resp. l elements of y). Symmetric Testing consists in finding test data that violate a given symmetry relation, i.e. finding θ such as for all η :

$$y = \theta.x \wedge p(y) \neq \eta.p(x)$$

Symmetries relations are generic properties and checking the correctness of programs in regards with these relations is a difficult task, likely undecidable in the general case[3]. However, there are circumstances when testing procedures can be used to check the correctness of properties against programs [3, 14, 16]. Hence, by using these procedures, it becomes possible to check that a given symmetry relation is satisfied by the program on a finite subset of its input space. Limitations of ST concern the weaknesses of symmetry relations to differentiate incorrect implementations from correct ones. In fact, there are lots of programs that satisfy a given symmetry relation and any incorrect implementation will not be necessary discovered by Symmetric Testing. Conversely, the approach does not report any spurious faults. In order to evaluate the fault revealing capacity of ST, we implemented it by using the four unit operations of the Java testing tool *Roast* [1] and we used it to reveal faults on several academic programs and on programs extracted from the third-party library **java.util.Collections.*** of the Java 2 plateform (std edition 1.4). These first experimental results show that ST is of particular interest when testing some of the "non-testable" programs.

The rest of the paper is organized as follows : section 2 presents the Group theoretic results as well as the necessary notations required to fully understand

[1] with the Euclidian algorithm for example

[2] $p(x)$ denotes the vector of values computed by the execution of p with x as input

[3] Although we are not aware of any proof of this, it can be hypothesized as a consequence of the undecidability of the halting problem

the paper. Section 3 details the principle of Symmetric Testing while section 4 reports the experimental results obtained with *Roast*. Related works are described in section 5 and finally section 6 indicates several perspectives to this work.

2 Group theory : notations and selected results

All the basic material on Group theory presented in this section is extracted from [15] and the JS Milne's lecture notes (available online `www.jmilne.org/math/CourseNotes`).

Definition 1 *(Group)*
A nonempty set G together with a composition law $\circ$ is a **group** *iff G satisfies the following axioms :*

- $\forall a, \forall b, \forall c \in G, a \circ (b \circ c) = (a \circ b) \circ c$ *(associativity)*
- $\exists e \in G$ *such as* $a \circ e = e \circ a = a \ \ \forall a \in G$ *(neutral)*
- $\forall a \in G, \exists a^{-1} \in G$ *such as* $a \circ a^{-1} = a^{-1} \circ a = e$ *(existence of an inverse)*

The symmetric group notion is the corner-stone of Symmetric Testing. Let E be a nonempty finite set of n distinct elements, the set S_E of bijective mappings from E to itself is called the **symmetric group** of E (say S_E acts on E). This symmetric group has exactly $n!$ elements, which are named permutations. It is clear that S_E is a group because it is closed and associative under $\circ$, identity is its neutral element and each permutation posses an inverse because the definition is restricted to bijective mappings only.

S_E can be identified with S_n : the symmetric group acting on $\{1, .., n\}$ as there is a trivial bijective relation (isomorphism) between S_E and S_n. A permutation in S_n is written : $\theta = \begin{pmatrix} 1 & .. & n \\ i(1) & .. & i(n) \end{pmatrix}$ where $i(1), .., i(n)$ denote the images of $1, .., n$ by the permutation θ. When a permutation of S_n is applied to a vector x of size n, we will write $\theta.x$ to denote the image of x by the permutation θ (say θ acts on x). For the sake of clarity, we will extend our notations to program compositions. If p is a program, $p \circ \theta$ will denote the application of p to the permutation θ of the elements of its input vector. Conversely, $\eta \circ p$ will denote the permutation η applied to the output vector of p.

All the permutations can be expressed by using only a few ones. Consider for example the permutation $\theta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$ of S_5, the same permutation can be captured by the following notation $\theta = (13)(245)$ where each pair of brackets denotes a specific permutation called an r_cycle. A permutation $(a_1 a_2 .. a_r)$ of S_n is an **r_cycle** iff it maps a_1 to a_2, a_2 to a_3, .. a_{r-1} to a_r and leave unchanged the other elements. A 2_cycle (written $(a_i a_j)$) is usually referred to as a transposition. A trivial property of transpositions is that they are their own inverse.

A subset X of elements of a finite group G is a set of **generators** iff every element of G can be written as a finite composition product of the element of X. G is said to be generated by X. For example, it is well-known that S_3 is generated by the two transpositions $\tau_1 = (12)$ and $\tau_2 = (23)$ because each one of the six elements of S_3 can be written with a finite composition product of these two transpositions. More generally, S_n is generated by all the transpositions, but also by the following subset of transpositions : $\{(12), (23), .., (n-1, n)\}$. In this paper we will use the following proposition, given here without a proof (that can be found in [15]).

Proposition 1 *(generators of S_n)*
The transposition $\tau = (12)$ and the n_cycle $\sigma = (12..n)$ together generate S_n.

It can be shown that S_n cannot be generated by less than two permutations. Hence $\{\tau, \sigma\}$ is a set of generators of minimum size.

A fundamental notion in Group theory is group homomorphism :

Definition 2 *(group homomorphism)*
A group homomorphism from a group G to a group G' over the same composition law $\circ$ is a map $\varphi : G \longrightarrow G'$ such that $\varphi(\theta \circ \theta') = \varphi(\theta) \circ \varphi(\theta')$.

Note that an isomorphism is simply a bijective homomorphism. As a consequence of this definition, the image of a group homomorphism from G to G' is a subgroup of G'. It is noted $Im(\varphi)$. Conversely, $Ker(\varphi)$ denotes the kernel of a group homomorphism, which is the set of permutations of G which are mapped to $id_{G'}$. $G/Ker(\varphi)$ denotes the group quotient of G by $Ker(\varphi)$, i.e. the set $\{g \circ h | g \in G, h \in Ker(\varphi)\}$. $Hom(G, G')$ denotes the set of group homomorphisms (in fact, it is also a Group). To end this review of the Group theoretic results that we need here, we give the fundamental theorem of group homomorphisms :

Proposition 2 *(isomorphism theorem)*
Let G and G' be two groups and φ be an element of $Hom(G, G')$, then φ factors into the composite of a surjection, an isomorphism $\bar{\varphi}$ and an injection :

$$\begin{array}{ccc} G & \xrightarrow{\varphi} & G' \\ {\scriptstyle surj}\downarrow & & \uparrow{\scriptstyle inj} \\ G/Ker(\varphi) & \xrightarrow{\bar{\varphi}} & Im(\varphi) \end{array}$$

3 Principle of Symmetric Testing

3.1 Symmetry relations

The idea behind Symmetric Testing is to exploit user-defined symmetries to automate the testing process of imperative programs. Lots of definitions of symmetry have been proposed in various contexts [15]. Some of them can be adapted for our purpose. We briefly discuss two possible choices that can be considered for program testing.

- **Symmetries over values**. A symmetry over values can be expressed as a relation between two program executions when there is a geometric relation (for example, an isometry) beetwen the two input points. As a trivial example, consider a program p which takes two integers as arguments and verifies $p(x, y) = p(-x, -y)$. In this case, the two input points are symmetrical w.r.t. the origin of the input space ;
- **Symmetries over variables**. A symmetry over variables can be viewed as a relation between two program executions where there is a permutation relation beetwen the input points. $p(x, y) = p(y, x)$ is the most simple example of such a symmetry.

Symmetries over values can easely be recognized for programs that compute a mathematical function given by a formula (based on arithmetic or trigonometric operations). In some cases, local symmetries over these operations may be aggregated to determine a global symmetry over the formula, such as in the formula $p(x, y) = sin(xy) - cos(y)$ which satisfies a trivial symmetry over values w.r.t. the origin. Nevertheless, in such a case the formula itself can be used to check the correctness of the computed outcome. Hence, these symmetries over values appear to be useless for our purpose. Conversely, symmetries over variables can be specified with a very few knowledge on the function being computed. Type informations are sometimes sufficient to see that a program has to satisfy a symmetry over variables. Further, they are properties that can be easely extracted from an informal specification. These are the reasons why we will focus on such symmetries in this paper. Formally speaking, a symmetry is defined as follows :

Definition 3 *(symmetry)*
Let p be a program over a domain D that takes n references as input and returns m references[4]*, and let S_n (resp. S_m) be the symmetric group over n (resp. m) elements, then a* **symmetry** *is a pair $< \theta, \eta >$ such as : $\theta \in S_n, \eta \in S_m$,*

$$y = \theta.x \implies p(y) = \eta.p(x) \quad \forall x, y \in D$$

Note that every program p satisfies at least the trivial symmetry $< id_{S_n}, id_{S_m} >$ because imperative programs are considered to be deterministics here (two executions with the same input give the same result). Some of the references of the input vector may be leaved unchanged by the permutation θ of a symmetry $< \theta, \eta >$. So, the vector of k exchanged input references involved in the symmetry is called the *permutable input* set[5] whereas the vector of l exchanged output references is called the *permutable output* set. Such symmetries can be grouped together by the mean of symmetry relations.

Definition 4 *(symmetry relation)*
Let p be a program over a domain D that has k per-

[4] As usual in imperative programming, the value of an input reference may be modified within the program and considered so as an output variable

[5] in Group theory, this is called the support of a permutation

mutable input data and l permutable outcomes, $\Psi_{k,l}$ is a **symmetry relation** *for p iff*

- $\Psi_{k,l} \in Hom(S_k, S_l)$ *(group homomorphism)* ,
- $\forall \theta \in S_k, \quad < \theta, \Psi_{k,l}(\theta) >$ *is a symmetry for p.*

The reason why symmetry relations are required to be group homomorphisms is based on our will to characterize the links between permutable outcomes. This will be made clearer in the following. Note that symmetry relations are very difficult to check when the number of permutable input data increases (because S_k contains $k!$ elements). It is important to see that $\Psi_{k,l}$ does not denote a unique symmetry relation, because there is no requirement over the mapping properties of the homomorphism. In fact, $\Psi_{k,l}$ is identified with a class of symmetry relations that are group homomorphisms in $Hom(S_k, S_l)$. Based on their formal definition, identifying such symmetry relations might appear to be difficult. Conversely, we argue that they can easily be specified by looking at the informal specification of programs, because they are often related to the type informations of program variables. Consider a program p taking an unordered set as argument, then we already know that p has to satisfy a symmetry relation because computing p with a permutation of the elements of the set does not modifiy the computed result. Numerous programs take unordered sets as arguments : consider sorting programs or graph-based programs just to name a few. Further, third-party libraries that contain lots of generic programs (for reusing purpose) have often to satisfy symmetry relations.

3.2 Examples

Consider the standard application programming interface specification of the `java.util.Collections.replaceAll` method given in Fig.1. If we consider the n_cycle σ (permutation $(12..n)$), then the method `replaceAll` has to satisfy a $< \sigma, \sigma >$ symmetry : let A (resp. B) be a vector of n symbolic references and A' (resp. B') be the resulting vector computed by invocation of `replaceAll` with the references $oldVal$ and $newVal$, then $B = \sigma.A \implies B' = \sigma.A'$. A and B are two permutable input sets whereas A' and B' are the permutable output sets. Further, it is clear that `replaceAll` has to satisfy the same symmetry for all $\theta \in S_n$. Hence, this Java method has to satisfy a $\Psi_{|A|,|A|}$ symmetry relation, where $|A|$ denotes the size of the abstract collection A. Finally, this group homomorphism is the identity of $Hom(S_{|A|}, S_{|A|})$, which is only one of the possible symmetry relations represented by $\Psi_{|A|,|A|}$. By looking at the `java.util.Collections` class which contains 19 distinct methods[6] among 37, we found that 12 methods have to satisfy at least one non-trivial symmetry relation. This class was selected because it consists of methods that operate on collections, which can be specialized on multiset or sequences of objects. As a consequence, the results given here should not be extrapolated for any other classes. Table 1 summarizes the symmetry relations found for these methods. Permutable input and output are indicated in the two central columns. The symbol Ret denotes the returned reference or value of the Java method.

```
public static boolean replaceAll(List A,
                                 Object oldVal,
                                 Object newVal,

  Replaces all occurrences of one specified value in a list with
  another. More formally, replaces with newVal each element e
  in A such that (oldVal==null ? e==null : oldVal.equals(e)).
  (This method has no effect on the size of the list.)
  Parameters:
    A - the list in which replacement is to occur.
    oldVal - the old value to be replaced.
    newVal - the new value with which oldVal is to be
    replaced.
  Returns:
    true if list contained one or more elements e such that
    (oldVal==null ?  e==null :  oldVal.equals(e)).
  Throws:
    UnsupportedOperationException - if the specified list or
    list-iterator does not support the set method.
```

Figure 1: API specification of `replaceAll`

The `max` and `min` have to satisfy a $\Psi_{|A|,1}$ symmetry relation because they return one of the elements of an unordered set, took as argument. Conversely, `fill, replaceAll, reverse, rotate, sort, shuffle,swap` have to satisfy a $\Psi_{|A|,|A|}$ symmetry relation because they modify ordered sequences or lists. In fact, these polymorphic functions have been extensively studied in the Functional Programming Community and the symmetry relations they have to satisfy can be derived from the well-known properties of their type [17]. `enumeration` has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation but $|Ret|$ is equal to $|A|$ in this case, hence the program has to satisfy the same symmetry relation than the other programs. Note that `shuffle` uses a random permutation of its permutable input data. Although the computed list cannot be easily predicted, the symmetry relation that `shuffle` has to satisfy is specified without any difficulties. `nCopies` has to satisfy a $\Psi_{1,|A|}$ symmetry relation, which can be interpreted as follows : whatever is the argument of `nCopies`, the outcome should be a vector of equal references. To complete this panorama, consider the `copy` method which aims at copying the values contained into a list (the source list) into another one (the destination list). The method requires the destination list to be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected and remain equal to their previous values. As a consequence, the method has to satisfy a $\Psi_{|A,|B|}$ symmetry relation where $|A|$ may not be equal to $|B|$. For example, copying $S = [1,2,3]$ into $D = [0,0,0,0,0]$ leads to $D' = [1,2,3,0,0]$. In the section 4, we give several examples of non-trivial symmetry relations.

[6] methods which have distinct specifications, and not only distinct interfaces

Signature of Java methods	Perm. in	Perm. out	Sym. rel.
`void copy(List B,List A)`	A	B	$\Psi_{\|A\|,\|B\|}$
`Enumeration enumeration(Collection A)`	C	Ret	$\Psi_{\|A\|,\|Ret\|}$
`void fill(List A,Object obj)`	A	A	$\Psi_{\|A\|,\|A\|}$
`Object max(Collection A)`	C	Ret	$\Psi_{\|A\|,1}$
`Object min(Collection A)`	C	Ret	$\Psi_{\|A\|,1}$
`List nCopies(int n, Object O)`	O	Ret	$\Psi_{1,\|Ret\|}$
`boolean replaceAll(List A, Object oldVal, Object newVal)`	A	A	$\Psi_{\|A\|,\|A\|}$
`void reverse(List A)`	A	A	$\Psi_{\|A\|,\|A\|}$
`void rotate(List A, int distance)`	A	A	$\Psi_{\|A\|,\|A\|}$
`void shuffle(List A)`	A	A	$\Psi_{\|A\|,\|A\|}$
`void sort(List A)`	A	A	$\Psi_{\|A\|,\|A\|}$
`void swap(List A, int i, int j)`	A	A	$\Psi_{\|A\|,\|A\|}$

Table 1: Examples of symmetry relations

3.3 Symmetric Testing

These symmetry relations can be used to seek a subclass of faults within an implementation. Formally speaking :

Definition 5 *Symmetric Testing*
Let p be a program and $\Psi_{k,l}$ be a symmetry relation for p, then Symmetric Testing aims at finding a triple $(x, p(x), p(\theta.x))$ such as $p(\theta.x) \neq \Psi_{k,l}(\theta).p(x)$

If found, such a triple $< x, p(x), p(\theta.x) >$ represents a counter-example for the symmetry relation. This shows that at least one of the two test data x and $\theta.x$ reveals a fault in p. So, given a set of test data and a symmetry relation, we get a naive procedure that can check whether program outcomes are incorrects. It is required to compute all the permutations of the permutable input of a vector x, to execute p with all these input data and to check whether the outcome vectors are equals to a permutation of the vector returned by $p(x)$. This principle can be illustrated by the following commutative diagram.

$$\begin{array}{ccc} x & \xrightarrow{\theta} & \theta.x \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle p} \\ p(x) & \xrightarrow[\Psi_{k,l}(\theta)]{} & \Psi_{k,l}(\theta).p(x) \end{array}$$

It is only a necessary condition for the correctness of p w.r.t. its specification because incorrect implementations of p may also satisfy the same symmetry relation. Note that this procedure is independant of the test set being used. In fact, it leaves the tester the possibility to use any automatic test data generators, because it is not required to produce an oracle for the expected outcomes. However, it should be recalled that the number of possible permutations in S_k is $k!$, leading to an impractical number of program calls whenever k increases. No hypothesis are made on the type of the permutable input data (Object references, integers, ...) but checking the equality between floating point values might be hazardous because programs that manipulate such variables depend strongly on the evaluation order of expressions. So, the equality relation between the computed results should be relaxed for these types.

3.4 Checking a given symmetry relation

We turn now on a more practical procedure that checks whether a program satisfies a given symmetry relation.

3.4.1 Reducing the number of permutations

In order to limit the number of program calls, we propose to check only two permutations when checking a symmetry relation. In fact, by using the proposition 1, we known that only two permutations are required to generate S_k. As a consequence, we get the following proposition :

Proposition 3 *Let p be a program and $\Psi_{k,l}$ be a symmetry relation for p, let $\tau = (12)$ and $\sigma = (12..k)$, then we have*

$$\begin{cases} p \circ \tau = \Psi_{k,l}(\tau) \circ p \\ p \circ \sigma = \Psi_{k,l}(\sigma) \circ p \end{cases} \iff p \circ \theta = \Psi_{k,l}(\theta) \circ p \ \forall \theta \in S_k$$

Proof: $\Longleftarrow$. $\tau \in S_k$ and $\sigma \in S_k$, hence taking $\theta = \tau$ and $\theta = \sigma$ yields the expected result.
$\Longrightarrow$. Let $\theta \in S_k$ be a permutation (distinct from τ or σ). By using proposition 1, θ can be written as a finite composition of the two permutations τ and σ. Let $\theta = \tau \circ \sigma \circ ...$ be the beginning of such a composition (taking any other chain does not change the proof) then $p \circ \tau \circ \sigma \circ ... = \Psi_{k,l}(\tau) \circ p \circ \sigma \circ ...$ by applying one of the two hypothesis and recalling that $\circ$ is associative. Further, it is possible to iterate on the composition chain : $p \circ \tau \circ \sigma \circ ... = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ p \circ ... = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ... \circ p$. This is repeated until the complete finite chain would have been processed. Finally, $\Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ...$ is equal to $\Psi_{k,l}(\theta)$ because $\Psi_{k,l}$ is a group homomorphism.

As a corollary, it is possible to characterize the subgroup of S_l, image of S_k by the homomorphism $\Psi_{k,l}$.

Proposition 4 *(Generators of $Im(\Psi_{k,l})$)*
$\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$ together generate the subgroup $Im(\Psi_{k,l})$.
Proof: $\Psi_{k,l}(\theta) = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ...$ for all $\theta \in S_k$ hence every permutation of $Im(\Psi_{k,l})$ can be written as a finite composition of $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$.

$Ker(\Psi_{k,l})$ denotes the set of permutations of S_k which leave the outcome of p unchanged. Because $G/Ker(\Psi_{k,l})$ is isomorphic to $Im(\Psi_{k,l})$ by the group isomorphism induced by $\Psi_{k,l}$, it is possible to determine precisely the mapping properties of $\Psi_{k,l}$ just by looking at the link between the two generators $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$, but this is outside the scope of this paper.

3.4.2 Checking only τ and σ

We will provide here a procedure to check whether a program p satisfies the two symmetries $< \tau, \Psi_{k,l}(\tau) >$ and $< \sigma, \Psi_{k,l}(\sigma) >$ over an input space D. In fact, it is required to show that $p(\tau.x) = \Psi_{k,l}(\tau).p(x)$ and $p(\sigma.x) = \Psi_{k,l}(\sigma).p(x)$ for all $x \in D$. To achieve such a goal by the mean of testing, we propose to use an local exhaustive test data generator [11]. However, other approaches, that make use of a (semi-)proving technique can be followed [14, 16] and are discussed in the section 5 of this paper. In general, input domains are infinites, as illustrated by the **replaceAll** method (Fig.1), which takes an unbounded list as first argument. In this case, a local exhaustive test data generator will enumerate all the possible lists until a selected size will be reached. So, any proof of symmetry relation satisfaction would be limited to the input domain being exhaustively explored. Fortunately, the limitation of the input space allows the approach to remain practical.

The keypoint of our approach is that we just have to know whether $p(\tau.x)$ and $p(\sigma.x)$ are permutations of $p(x)$. As previously said, a precise knowledge of $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$ is not required here because $\Psi_{k,l}$ represents an entire class of symmetries. The procedure shown in Fig.2 takes a program p and the input space D as arguments and returns the first found triple $< x, p(x), p(\theta.x) >$ that violates the symmetry relation, among the portion of the input space being explored. If such a counter-example cannot be found, this proves that p satisfies not only the two selected symmetries but also all the permutations of S_k of the input vector in the input space D. Note that some test data are not required to be examined : test data of the form $x = (v, v, .., v)$ can be eliminated from D because any permutation will leave x unchanged.


```
while(D ≠ ∅) {
    pick up x ∈ D,
    D := D \ {x};
    if( p(τ.x) is not a permutation of p(x) )
        return < x, r, p(τ.x) >
    if( p(σ.x) is not a permutation of p(x))
        return < x, r, p(σ.x) >
    }
    return("Check complete");
```

Figure 2: A procedure for Symmetric Testing

Note also that non-permutable input data may be leaved constants because these input data do not play any role in the symmetry relation. However, by doing these, we restrict the proof when the procedure explores the complete domain.

3.5 Discussion

As expected, terminaison of the previous procedure cannot be guaranted. Although the input space of the program p is required to be finite, nothing prevents p to iterate infinitely when computing $p(x)$,$p(\tau.x)$ or $p(\sigma.x)$ and no general procedure can be used to decide the termination of p.

Under the strong hypothesis that p halts on all test data of its finite input space, ST is guaranted either to find a counter-example of the symmetry relation if there exists one, or to show that p satisfies the symmetry relation. However, the input space of p may have to be fully explored in the worst case. Let $D_1 \times D_2 \times .. \times D_n$ be the finite input space of p and let d be the number of elements of the greatest domain D_i, then the procedure given in Fig.2 will have to enumerate $O(d^n)$ points in the worst case. From a practical point of view, it is crucial to maintain d and n as smallest as possible by limiting the size of the domains of permutable input data. Note also that the number of program calls is $O(d^n)$ by using the procedure of Fig.2 whereas it would have been $O(n!\ d^n)$ in the worst case by using the naive procedure that we first introduced.

Another limitation comes from the difficulty to establish that an extracted symmetry relation is actually a group homomorphism, for some programs. Consider the method `Vector min_nb(Vector A, int nb)` which computes a vector of nb minimum integer values extracted from A, given in Fig.3. We can guess that this program has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation. However, the vector returned by the program is not sorted and its elements order depends strongly on the algorithm used. Further, it is difficult to verify at hand that $\Psi_{|A|,|Ret|}(\theta_1 \circ \theta_2) = \Psi_{|A|,|Ret|}(\theta_1) \circ \Psi_{|A|,|Ret|}(\theta_2)$ for all θ_1, θ_2. An approach for this problem would be to use symmetry relations over compositions of programs. For example here, composing `min_nb` with a sorting program of the resulting vector yields a $\Psi_{|A|,1}$ symmetry relation for the composition.

```
public static Vector min_nb(Vector V, int nb) {
  if( (V == null) ) return null;

  int k = ((V.size() < nb) ? V.size() : nb);
  int j, i = 0 ;
  Vector R = new Vector();
  Integer max_R, cur_V;
  Collections c = null;

  while( i < k ) {
    R.addElement(V.elementAt(i));
    i ++;
  }

  while(i < V.size()){
    j = 0;
    while(j < k){ // k is the size of R
      max_R = (Integer)c.max(R)   ; // max value of R
      cur_V = (Integer)V.elementAt(i); // current value of V

      if(max_R.intValue() > cur_V.intValue()){
        R.setElementAt(cur_V, R.indexOf(max_R));
        break;
      }
      j++;
    }
    i++;
  }
  return(R);
}
```

Figure 3: The `min_nb` program

4 Experimental results

4.1 Implementation

We implemented ST with the help of the primitive operations of the Java unit testing tool *Roast* [1]. The tool includes four unit operations (generate, filter, execute, and check) designed 1) to automate the generation of test tuples, 2) to filter some the generated tuples, 3) to execute the program under test with the selected tuples and 4) to check the computed outcomes. *Roast* provides several test data generators such as a boundary-value generator and a Cartesian product generator. We used only the latter to implement our local exhaustive test data generator. Roast supports test templates (Perl macros) to compare actual outcomes to predicted ones. We used these templates in combination with our Java methods to check whether a computed outcome was a permutation of another one.

4.2 Experiments with ST

The goal was to study the capacity of ST to reveal faults in programs and to find circumstances where ST checks that a given symmetry relation is satisfied. The experiment was performed on classical academic programs, where faults were injected by mutation.

Programs. Six programs were selected among which three were implemented and three came from the **java.lang.Collections** class : `replaceAll`, `sort` and `copy`. We implemented the `min_nb` method (given in Fig.3), the `GetMid` program (given in [14]) intended to compute the median of three integers, and the well–known triangle classification program `trityp` [18]. This program takes three integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene (sca), isocele (iso), equilateral (equ) or illegal (illeg). To limit the size of the search space, we considered every input integer to belong to a range of 100 values (ranging from 0 to 99) for `GetMid` and `trityp`. For `min_nb`, `replaceAll`, **copy**, and **sort** we considered lists to contains at most 4 integers ranging from 0 to 19.

Mutants. Four mutants were created for `min_nb` and `GetMid`. `GetMid_1` and `min_nb_1` are versions of the original programs where two statements are removed. The first mutant has been studied in [14] because it contains a "missing path error" fault, which is considered as a difficult fault to reveal. The mutation of relational operators in `GetMid_2` and `min_nb_2` leads to the creation of infeasible paths (at least for the first program). Finally, thirty three mutants were manually created for `trityp`. The strategy used to create the mutants was to exchange operators, values or variables in a systematic manner. Equivalents mutants[7] have been removed from the set of experiments, because they cannot be revealed by the mean of testing [18]. All the mutants are available at the url `www.irisa.fr/lande/gotlieb`

Symmetry relations. The results of `GetMid` and `trityp` must be invariant to every permutation of their three input values, leading to a $\Psi_{3,1}$ symmetry relation. As previously said, `min_nb` has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation. Finally, Table 1 contains the expected symmetry relations for `replaceAll`, `copy` and `sort`.

4.3 Experimental results and Analysis

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`min_nb_1`	vec:[1,1,0], nb:2	**[0,0]**	[1,0]	**[0,0]**	6.8
`min_nb_2`	vec:[1,0], nb:1	[0]	**[1]**	[0]	4
`min_nb`	3367220 test data				57

Table 2: Results for `min_nb`

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`GetMid_1`	(1,1,0)	**0**	1	**0**	0.1
`GetMid_2`	(1,0,0)	0	**1**	0	0.1
`GetMid`	999900 test data				9.4

Table 3: Results for `GetMid`

[7] programs which compute the same outcome as the original program although a mutation operator has been applied

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`trityp_1`	(3,1,1)	**iso**	illeg	illeg	0.2
`trityp_2`	not found				9.6
`trityp_3`	(2,1,1)	illeg	**iso**	**iso**	0.2
`trityp_4`	(2,1,1)	**equ**	**sca**	**iso**	0.2
`trityp_5`	(2,1,1)	**iso**	illeg	illeg	0.2
`trityp_6`	(2,1,1)	illeg	illeg	**iso**	0.2
`trityp_7`	(2,1,1)	illeg	illeg	**iso**	0.2
`trityp_8`	(2,2,1)	**equ**	iso	**equ**	0.2
`trityp_9`	not found				9.3
`trityp_10`	(2,1,1)	**equ**	illeg	illeg	0.2
`trityp_11`	not found				9.2
`trityp_12`	(2,2,1)	**illeg**	iso	**illeg**	0.2
`trityp_13`	not found				9.2
`trityp_14`	not found				9.6
`trityp_15`	not found				9.4
`trityp_16`	not found				10
`trityp_17`	(3,2,1)	**sca**	illeg	**sca**	0.2
`trityp_18`	not found				9.6
`trityp_19`	(2,2,1)	**sca**	iso	**sca**	0.2
`trityp_20`	(3,2,1)	**sca**	illeg	illeg	0.2
`trityp_21`	(3,2,1)	illeg	**sca**	illeg	0.2
`trityp_22`	(2,1,1)	illeg	illeg	**equ**	0.2
`trityp_23`	(2,1,1)	illeg	**equ**	**equ**	0.2
`trityp_24`	(2,1,1)	illeg	illeg	**equ**	0.2
`trityp_25`	(2,1,1)	**equ**	**iso**	illeg	0.2
`trityp_26`	(2,1,1)	illeg	**iso**	illeg	0.2
`trityp_27`	(2,1,1)	**equ**	illeg	illeg	0.2
`trityp_28`	(2,1,1)	illeg	**iso**	illeg	0.2
`trityp_29`	(2,1,1)	**equ**	**iso**	**iso**	0.2
`trityp_30`	not found				9.9
`trityp_31`	(1,1,0)	illeg	**iso**	illeg	0.2
`trityp_32`	(1,0,1)	illeg	**iso**	**iso**	0.1
`trityp_33`	not found				10
`trityp`	999900 test data tried				9.6

Table 4: Results for `trityp`

All the computations were performed on a 1.8Ghz Pentium 4 personal computer by using the version 1.3 of *Roast*.

4.3.1 Revealling faults with ST

We first applied ST to reveal faults among the mutants of each program. The results are given in tables 2, 3, and 4. If found, a test datum x that violates the symmetry relation is given. The values of $p(x)$, $p(\tau.x)$ and $p(\sigma.x)$ are given in each of three interiors columns of the tables. In the case where a test datum is found , the mutant is said to be killed by the symmetry relation. Incorrect results of the program p are noted with boldface to facilitate the results interpretation, but this was determined manually. For each relation, the CPU time spent to find the solution is given (including time spent garbage collecting or in system calls) in the last column.

Test data are found for killing the two mutants of `min_nb` and `GetMid`. This illustrates the capacity of ST to reveal two difficult class of faults (missing path error and infeasible path) on thess programs. Among the 33 mutants of `trityp`, 10 were not detected as faulty versions (programs `trityp` _2,_9,_11,_13,_14,_15,_16,_18,_30, _33) by ST. Studying these programs leads to see that they are equivalent to the correct version of `trityp` in the following sense : both the mutant and the correct version satisfies the same symmetry relation. For example, the mutant `trityp_9` cannot be detected by ST because the fault has been introduced into a statement only reached by a sequence of three equal integers, which is invariant to permutation and in fact not even tried by the local exhaustive generator. In some cases, the $p(x)$, $p(\sigma.x)$ and $p(\tau.x)$ are all incorrects (mutants _4 and _29). This illustrates a situation where a fault injected in the program yields to modify every computed outcome. Fortunately, this breaks also the symmetry relation that the program has to satisfy.

4.3.2 Checking that a program satisfies a given symmetry relation

Checking that the symmetry relations are satisfied by the correct versions of `min_nb`, `GetMid`, and `trityp` yields to the results shown in the last row of table 2, 3 and 4. As the size of the search space was arbitrarely limited, the proof is only valid on a small part of the input space. Nevertheless, we prefered to compromize the size of the input space rather than the time spent to search a counter-example. Although these proofs are done in restricted cases, they form a valuable step toward program correctness because the checking procedure is completely automated for these programs. Finally, we applied ST to check symmetry relations for the `copy`, `sort`, `replaceAll` methods of the class `java.lang.Collections`. The results are given in table 5 and show that the three methods satisfy their symmetry relation among the restricted input space.

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`copy`	168361 test data				14.4
`sort`	168361 test data				5.4
`replaceAll`	67344400 test data				38240.5

Table 5: Results for `sort` `copy`, `replaceAll`

5 Related work

The idea of checking the computed results of a program by using several input data is not new. Ammann and Knight [19] described the data diversity approach which aims at executing the same program, on a related set of input points. To check the computed outcomes, a voting procedure is used as an acceptance test. As claimed by the authors, the re-expression algorithms, used to generate the input data in relation within an expression, must be tailored to the application at hand. Blum and Kannan [20] proposed the concept of *program checker*, which is a program able to play the role of a probabilistic oracle, under a set of restrictions. As an example, the authors considered the graph isomorphism problem and they provide a program checker which checks that a graph G' resulting from a random permutation of a graph G is isomorphic to a graph H only if G is isomorphic to H. Other program checkers that make use of mathematical properties are designed for programs that compute *gcd*,the matrix rank or programs that sort data. Conversely to these works, our approach focus on generic relations that can be easily extracted from an informal specification. The symmetry relations that are considered in this paper are general and do apply to programs not restricted to compute mathematical functions.

More recently, Chen et al. proposed in [21] to use existing relations over the input data and the computed outcomes to eliminate faulty programs, in a framework called Metamorphic Testing. It is proposed in [14] to use global symbolic evaluation techniques to prove that an implementation satisfies a given metamorphic relation for all input data. The technique yields to enumerate the paths of the program and to evaluate the statements along each path by replacing variables by symbolic values. Iterations are treated by a complex and manual loop analysis. The procedure requires to compare several sets of constraints extracted from the program, which are manually simplified on two example programs. One of them is the `GetMid` program that we used in our experiments. For this program, the same symmetry relation we gave, is provided and only a few permutations (transpositions only) are used to check the relation. However, their approach is only manual and seems difficult to be automated. In a previous work [16], we attempted to automate the generation of input data that violate a given metamorphic relation, by using Constraint Logic Programming techniques. During this work, symmetry relations appear to be of a great interest because of their simlicity and genericity. Furthermore, it appears that a formal basis were required to generalize the ideas based on the use of symmetry.

6 Perspectives

In this paper, we have introduced a new software testing paradigm called Symmetric Testing which aims at finding test data that violate a given symmetry relation. Group theoretic results are used to give a formal basis to this paradigm. In particular, a formal definition of symmetry relation is introduced and we have given a practical procedure for applying Symmetric Testing on imperative programs. However, the limits of our automated approach of Symmetric Testing have been identified. We forsee to replace the local exhaustive test data generator by a a constraint-based test data generator which makes use of constraints extracted from the source code. In this approach, constraints are used as relational expressions between input and output symbolic variables and allow us to prove properties about the program and its test data. We believe that symmetry relations would be easily expressed as program properties into such a framework. Further, program composition appear to be an interesting perspective to take into account programs for which it is not easy to specify symmetry relations. This would allow to specify symmetry relations over finite sequence of method invocations, providing so a way to test programs at the integration level.

Acknowledgment

References

[1] N. Daley, D. Hoffman, and P. Strooper. A framework for table driven testing of Java classes. *Soft. Prac. and Exper.*, 32(5):465–493, April 2002.

[2] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Soft. Eng.*, SE-2(3):215–222, September 1976.

[3] R. Boyer, B. Elspas, and K. Levitt. SELECT - A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, June 1975.

[4] Bogdan Korel. Automated Software Test Data Generation. *IEEE Trans. on Soft. Eng.*, 16(8):870–879, Jul. 1990.

[5] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA). Soft. Eng. Notes*,23(2):53-62, 1998.

[6] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Soft. Eng. Jour.*, 6(6):387–405, 1991.

[7] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Trans. on Soft. Eng.*, 20(5):353–363, May 1994.

[8] J. M. Voas. Certifying off-the-shelf software components. *Computer*, 31(6):53–59, June 1998.

[9] Joe Duran and Simeon Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10(4):438–444, Jul. 1984.

[10] Glenford J. Myers. *The Art of Software Testing*. John Wiley, New York, 1979.

[11] T. Wood, K. Miller, and R. E. Noonan. Local exhaustive testing: a software reliability tool. In *Proc. of the Southeast regional conf.*, pages 77–84. ACM Press, 1992.

[12] Elaine Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, 1982.

[13] P. Devanbu and S. G. Stubblebine. Cryptographic verification of test coverage claims. In M. Jazayeri and H. Schauer, editors, *Proc. of the European Soft. Eng. Conf. (ESEC/FSE)*, pages 395–413. LNCS 1013, September 1997.

[14] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[15] M. A. Armstrong. *Groups and Symmetry (Undergraduate Texts in Mathematics)*. Springer Verlag, second edition, 1988.

[16] A. Gotlieb and B. Botella. Automated metamorphic testing. PI 1516, IRISA Tech. Report - Rennes, France, 2003.

[17] P. Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, September 1989.

[18] R. A. Demillo and A. J. Offut. Constraint-Based Automatic Test Data Generation. *IEEE Trans. on Soft. Eng.*, 17(9):900–910, Sep. 1991.

[19] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. on Computers*, 37(4):418–425, 1988.

[20] M. Blum and S. Kannan. Designing programs that check their work. *Jour. of the Assoc. for Computing Machinery*, 42(1):269–291, January 1995.

[21] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Fault-based testing in the absence of an oracle. In *IEEE Int. Comp. Soft. and App. Conf. (COMPSAC)*, pages 172–178, 2001.

Chapter 5

Modélisation à contraintes des programmes avec pointeurs

Un développement important de nos travaux en génération automatique de tests à base de contraintes a concerné le traitement des pointeurs en C. En effet, le langage C offre de multiples possibilités de traitement des données grâce aux pointeurs et aux variables de type pointeur. Cependant, l'analyse et la compréhension des programmes qui manipulent les pointeurs est délicate car l'accès aux données pointées peut dépendre du flot d'exécution. Ce sujet est difficile pour les outils de test à base de contraintes car il réclame la définition d'un modèle mémoire qui impacte l'ensemble du traitement des contraintes. Nous présentons dans ce chapitre deux modèles à contraintes distincts, proposés pour la modélisation des calculs sur les pointeurs en test à base de contraintes. Le premier modèle dérive de travaux que nous avions initiés en 2000 [Gotlieb 00a] et que nous avons développés jusqu'à obtenir des résultats expérimentaux satisfaisants [Gotlieb 05b, Gotlieb 05a, Gotlieb 07]. Ce modèle est efficace mais aussi limité puisqu'il ne permet pas la prise en compte des structures de données dynamiques. A l'inverse, le second modèle, développé en parallèle [Gotlieb 06b, Charreteur 07] est plus général et permet de traiter un sous-ensemble du langage C beaucoup plus étendu, incluant l'allocation dynamique de mémoire et la déallocation. Il est cependant moins efficace que le premier, c'est pourquoi il est opportun de présenter les deux modèles, tout en sachant que ce thème de recherche n'est pas clos [Botella 09].

Ces deux modèles sont largement indépendants dans la mesure où l'un n'est pas un raffinement de l'autre ; ils correspondent plutôt à deux manières distinctes de traiter le problème de la synonimie dans les programmes à base de pointeurs. Ce problème concerne la possibilité d'accéder à la même case mémoire, au travers d'expressions syntaxiques différentes. Le premier modèle s'appuie sur un précalcul statique des relations de pointage, tandis que le second découvre ces relations lors de la résolution du système de contraintes. Le premier modèle est très efficace comme en témoigne nos résultats expérimentaux, mais restreint puisqu'il

ne permet pas de référencer des zones anonymes de la mémoire. En effet, le premier modèle ne peut contraindre que des zones mémoire auxquelles un nom du programme est attaché, ce qui exclu l'allocation dynamique de mémoire. Comme chaque pointeur (constante ou variable) est modélisé par une variable logique et une sur-approximation des relations de pointage est estimée avant résolution du système de contraintes, la génération de données de test est très efficace. A l'inverse, le modèle à contraintes du second article est plus général puisque chaque instruction qui modifie la mémoire est modélisée comme une relation entre deux états-mémoire abstraits. Ainsi, l'allocation dynamique de mémoire est représentable. Ce deuxième modèle représente les instructions comme des contraintes ensemblistes, où les ensembles seraient non nécessairement bornés [Charreteur 08]. Ce modèle mémoire ouvre des perspectives en matière de modélisation à contraintes des programmes impératifs qui manipulent les pointeurs, mais également des programmes orientées-objet où la notion de référence et l'allocation dynamique d'objet est omniprésente. Ce point fait l'objet du chapitre suivant.

A. Gotlieb, T. Denmat, and B. Botella. *Goal-oriented test data generation for pointer programs*. **Information and Software Technology, 49(9-10):1030–1044, Sep. 2007.**

Goal-oriented test data generation for pointer programs

Arnaud Gotlieb [a,*], Tristan Denmat [a], Bernard Botella [b].

[a] *IRISA / INRIA Campus Beaulieu 35042 Rennes Cedex, France*

[b] *THALES AEROSPACE 78851 Elancourt Cedex, France*

Abstract

Automatic test data generation leads to the identification of input values on which a selected path or a selected branch is executed within a program (path-oriented vs goal-oriented methods). In both cases, several approaches based on constraint solving exist, but in the presence of pointer variables only path-oriented methods have been proposed. Pointers are responsible for the existence of conditional aliasing problems that usually provoke the failure of the goal-oriented test data generation process. In this paper, we propose an overall constraint-based method that exploits the results of an intraprocedural points-to analysis and provides two specific constraint combinators for automatically generating goal-oriented test data. This approach correctly handles multi-levels stack-directed pointers that are mainly used in C programs. The method has been fully implemented in the test data generation tool INKA and first experiences in applying it to a variety of existing programs are presented.

Key words: Goal-oriented test data generation, Constraint Logic Programming, Static Single Assignment form, pointer variables

1 Introduction

Goal–oriented test data generation leads to identify input values on which a selected branch in a program is executed. The presence of pointer variables introduces technical difficulties making the extension of current goal–oriented test data generation methods a challenging task.

* Corresponding author
Email address: `Arnaud.Gotlieb@irisa.fr` (Arnaud Gotlieb).


```
        int f(int i, int j, int c) {
1.      int* p = &j;
2.      if (c == 1)
3.              p = &i;
4.      i = 0 ;
5.      *p = 1;
6.      if (i != 0)
7.              ...
```

Figure 1. A conditional aliasing problem

What is exactly the problem? In imperative programs, a dereferenced pointer and a variable may refer to the same (stack-based) memory location at some program point (a.k.a. an aliasing problem). This can be due either to a statement in the code where a pointer variable is assigned the address of another variable or to a relation over the pointer input values of a function. In the former case, the dependence may be conditioned by the control flow: a dereferenced pointer may be aliased with a variable only if some conditions that depend on the flow are satisfied. We call this situation a conditional aliasing problem. For example in the C code of Fig.1, `*p` may be aliased to `i` or `j` at statement 5. Consider the problem of generating a test datum that reaches branch 6-7. If the assignment of statement 5 is considered to have no effect on variable i, then the branch 6-7 will be declared as unreachable by an automated test data generator as $i = 0$ and $i \neq 0$ are contradictory. However, this is incorrect if the flow passes through statement 3 as, in this case, p points to i and then i is assigned to 1 at statement 5, which satisfies the decision of branch 6-7. On the contrary, if statement 5 is considered to be able to modify any pointed variable in the program, then the test data generation process suspends as it cannot decide whether $i \neq 0$ is satisfied or not. In this example, it is worth noticing that reaching branch 6-7 requires $c = 1$ to be satisfied, which is a necessary (and sufficent) condition to solve this problem. Note that when a path is selected first, the pointing relations are all known and such conditional aliasing problems are trivially handled, but if the selected path is infeasible then this must be demonstrated before switching to another choice and carrying on the process.

Contributions. In this paper, we propose to extend an existing constraint-based goal-oriented method [1,2] to take into account conditional pointer aliasing problems. Firstly, we propose the definition of a *Static Single Assignment form* (SSA)[3] in the presence of pointer variables. This SSA form integrates the results of an intraprocedural flow–sensitive pointer analysis in order to reveal the hidden definitions realized by dereferenced pointers. In the example of Fig.1, such an analysis says that `p` points either to `i` or `j` at statement 5. Secondly, we proposed the definition of two specific constraint combinators that model the existing relations between dereferenced pointers and variables.

In order to reach branch 6-7 in the example, these combinators correctly entail that $c = 1$. In this paper, we formally present the operational semantics of these combinators under the form of guarded constraints, and we detail our implementation while providing preliminary experimental results. As a consequence, this paper introduces an overall goal-oriented method able to correctly deal with programs that contain conditional pointer aliasing problems and (multi-level) pointers toward stack memory locations. Note however that our approach suffers from the following restriction: it cannot handle accurately dynamic allocated structures when dynamic allocation is placed within the body of an unbounded loop. This restriction should be minimized by the fact that in most critical systems (military and civil avionic, railway and automotive industries, etc.) dynamic allocation is prohibited [4].

Outline of the paper. In Section 2, the background on our constraint-based technique is recalled. Section 3 gives an overview of the approach on a motivating example. Section 4 details the Pointer–SSA form while section 5 presents two specific combinators used to model pointer use and definition. Section 6 reports on the experimental results we obtained with our implementation in the test data generator INKA and Section 7 discusses related work. Finally, Section 8 recalls the contributions of the paper and indicates several perspectives.

2 Background

Constraint-based test data generation. Originally introduced by DeMillo and Offut in the context of mutation testing [5], Constraint-Based Test data generation (CBT) aims at exploiting constraint satisfaction techniques to generate test data able to reach a selected branch in a program under test. The method builds a constraint system associated to a given branch and then tries to solve the system by using domain reduction techniques. Several tools support CBT: the Godzilla system [6,7] exploits a dynamic domain reduction technique to reach mutation operators in Fortran programs, INKA [1,2] uses Static Single Assignment form and Constraint Logic Programming (CLP) techniques over finite domains to generate test data for the structural coverage of C programs, and ATGen [8] exploits symbolic execution for structural coverage of Spark ADA programs. Recently, INKA has been improved to handle floating-point computations [9], function calls and structures [10]. Although these CBT's implementations have proved to be useful to address non-trivial academic and industrial test data generation problems (including loops, arrays, structures, bitwise operations and so on) it is worth noticing that none of them was able to deal correctly with pointer aliasing problems.

In the CBT approach of [1,2], the selection of a branch in the C function leads

to set up a Constraint Logic Programming request built with the control-dependencies [11]. Control-dependencies are decisions that must be evaluated to "true" to reach a selected branch. In well–structured programs (without goto statement), they can easily be computed [12], even if they must be determined dynamically for the loop statements. In the example of Fig.1, the control-dependency associated to branch 6-7 is just $i \neq 0$. In addition, type declarations are translated into domain constraints. For example, using a signed 32-bits integer x as an input variable leads to set up the following domain constraint: $X \in -2^{31}..2^{31}-1$. The last phase of the test data generation process consists in solving the resulting CLP request by using the techniques described in section 2.2. As the semantics of the program is modeled faithfully, a solution of the CLP request is correctly interpreted as a test datum that reaches the selected branch. In cases where the solving process shows that there is no solution, then the selected branch is declared unreachable. This approach has been implemented in the INKA tool [13] and evaluated on a set of academic and reasonably-sized industrial problems [2]. In [14], we also proposed to use this framework to generate test data that violate high-level properties called metamorphic–relations [15].

Our approach is based on the use of Static Single Assignment form [3] and Constraint Logic Programming over finite domains [16]. It is worth noticing that generating test data for reaching a given decision within a program requires to solve the problem of destructive assignment in imperative programming, e.g. to convert $i = i + 1$ into a relation of the form $i_2 = i_1 + 1$ where i_2, i_1 corresponds to distinct variables. In our approach, we proposed to use SSA for such a task several years ago [1].

2.1 SSA form

The SSA form is a semantically equivalent version of a program that respects the following principle : each variable has a unique definition and every use of this variable is reached by the definition. Every program can be transformed into SSA by renaming the uses and definitions of the variables. For example $i = i + 1; j = j * i$ is transformed into $i_2 = i_1 + 1; j_2 = j_1 * i_2$. At the junction nodes of the control structures, SSA introduces special assignments called ϕ-functions, to merge several definitions of the same variable : $v_3 = \phi(v_1, v_2)$ assigns the value of v_1 in v_3 if the flow comes from the first branch of the decision, the value of v_2 otherwise. SSA has been used in several applications area such as optimizing compilers, automatic parallelization, static analysis and automatic test data generation [1,2,17]. For convenience throughout the paper, we will write a list of successive ϕ-functions with a single statement over vectors of variables : $x_2 = \phi(x_1, x_0), .., z_2 = \phi(z_1, z_0) \Longleftrightarrow \vec{v_2} = \phi(\vec{v_1}, \vec{v_0})$ where

$\vec{v_i}$ denotes a vector $\begin{bmatrix} x_i \\ ... \\ z_i \end{bmatrix}$. SSA provides special expressions to handle arrays : $access(a, k)$ which evaluates to the kth element of a, and $update(a_0, j, v)$ which evaluates to an array a_1 which has the same size and the same elements as a_0, except for j where value is v.

2.2 The CLP(FD) framework

Following the definitions of [16], a *CLP(FD) program* is a set of clauses of the form $A :- B$ where A is a user-defined constraint and B is a a sequence of either primitive constraints or combinators calls[1] . Such a sequence is called a *query*. *Primitive constraints* are built with variables, domains, arithmetical operators in $\{ +, -, \times, \setminus \}$ and relations $\{ >, \geq, =, \neq, \leq, < \}$. In general, variables of the CLP(FD) program (called *FD_variables*) take their values into a non-empty finite set of integers.

Combinators are language constructs expressing a high-level relation between other constraints. They can be either built-in or user-defined constraint depending on the CLP(FD) interpreter that is used. For example, the combinator `element`(I, L, V) is built-in in the CLP(FD) library of Sicstus Prolog [18] : it holds if V is the Ith element in the list L of FD_variables.

When considered for solving, a CLP(FD) query leads to build dynamically a *constraint system*, which is made of variables, domains and constraints. Note that constraint solving over finite domains is NP-hard hence some approximations are usually performed before going into a brute force approach. Informally speaking, the solving process of a constraint system is based on 1) a constraint propagation mechanism which makes use of the constraints to prune the search space, 2) a constraint entailment mechanism which tries to infer new constraints from existing ones, 3) a labeling process which explores the search tree by making assumptions in order to find solutions to the constraint system.

Constraint propagation. During this process, primitive constraints and combinators are incrementally introduced into a propagation queue. An iterative algorithm considers each constraint one by one into this queue by filtering the domains of FD_variables of their inconsistent values. Filtering algorithms consider usually only the bounds of the domains. When the domain

[1] Throughout the paper, we will use the Prolog syntax for CLP(FD) programs

of a FD_variable is pruned then the algorithm reintroduces in the queue all the constraints where this FD_variable appears (awaked constraints) to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fixpoint). When selected in the propagation queue, each constraint is added into a *constraint–store* which memorizes all the considered constraints. The constraint–store is contradictory if the domain of at least one FD_variable becomes empty during the propagation.

Constraint entailment. Some constraints are designed to include conditional information. These constraints are defined with the help of *guarded–constraints*, noted $C_1 \longrightarrow C_2$. During constraint propagation, if C_1 is entailed then C_2 is introduced into the propagation queue, allowing so to dynamically enrich the constraint system. When the constraint C_1 is disentailed then the guarded–constraint $C_1 \longrightarrow C_2$ is just removed from the constraint–store. Otherwise, the guarded–constraint is suspended until being awaked by the constraint propagation mechanism.

Variable labeling. As it is usually the case with finite domain constraint solvers, constraint propagation does not ensure that the set of constraints is satisfiable when a fixpoint is reached. One must resort to enumerate to get particular solutions. This labeling procedure tries to give a value to every FD_variable one by one and propagates throughout the constraint system. This is done recursively until all the FD_variables are instantiated. It is noted `labeling`$([X_1, .., X_n])$ where $X_1, .., X_n$ is a n–tuple of FD_variables to instantiate. If this valuation leads to a contradiction then the procedure backtracks to other possible values. The valuation is done according to some strategies of choice of FD_variables and values. A simple one consists in selecting the minimum value of the domain of the first unbounded FD_variable. Of course other more sophisticated strategies can be used.

2.3 Combinators for program analysis

Based on the CLP(FD) framework, our approach consists in translating every statement into a constraint or a specific combinator. In [1], we introduced such combinators for the conditional and the iterative statements. Let us recall in this subsection the semantics of these combinators.

Conditional statement. The conditional statement is treated with the (user–defined) combinator `ite/6`[2] . Arguments of `ite/6` are made of the vari-

[2] where /6 denotes the arity of the combinator

Original C code	Pointer–SSA form	CLP(FD) program *where* $\&j = 21$ and $\&k = 22$
int foo(**int** i){ **int** j,k,r,$*p$;	**int** foo(**int** i){ **int** j,k,r,$*p$;	**foo**$(I, J_4, ...)$:– $I \in -2^{31}..2^{31}-1$
1. $j = 0$; 2. $k = 0$; 3. $p = \&j$;	$j_1 = 0$; $k_1 = 0$; $p_1 = \&j$;	$J_1 = 0$, $K_1 = 0$, $P_1 = 21$,
4. **if** ($i < 6$)	**if** ($i < 6$)	ite$(I < 6, \begin{bmatrix} J_2 \\ P_1 \end{bmatrix}, \begin{bmatrix} J_1 \\ P_2 \end{bmatrix}, \begin{bmatrix} J_3 \\ P_3 \end{bmatrix},$
5. $j = 2$; **else**	$j_2 = 2$; **else**	$J_2 = 2$,
6. $p = \&k$;	$p_2 = \&k$; $\begin{bmatrix} j_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} j_2 \\ p_1 \end{bmatrix}, \begin{bmatrix} j_1 \\ p_2 \end{bmatrix})$;	$P_2 = 22)$
7. $r = *p$;	$r_1 = \phi_u(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$	$R_1 = \Phi_u(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$,
8. $*p = r * i$;	$\begin{bmatrix} j_4 \\ k_2 \end{bmatrix} = \phi_d(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, r_1 * i, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$;	$R_2 = R_1 * I$, $\begin{bmatrix} J_4 \\ K_2 \end{bmatrix} = \Phi_d(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, R_2, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$,
9. **if** ($j > 8$) 10. ...	**if** ($j_4 > 8$) ...	ite$(J_4 > 8, ...)$,

Figure 2. An example, its PSSA form and the generated CLP(FD) program

ables that appear in the ϕ-functions and the constraints generated from the then– and the else– parts of the statement. Other combinators may be nested in the arguments of `ite/6`. An SSA **if_else** statement: **if** (*exp*) **{** *stmt* **}** **else** **{** *stmt* **}** $\vec{v_2} = \phi(\vec{v_0}, \vec{v_1})$
is converted into ite$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$ where c is a primitive–constraint generated by the analysis of *exp*, and C_{Then} (resp. C_{Else}) is a set of constraints generated by the analysis of the then–part (resp. else–part).

Definition 1 `ite/6`

Declarative semantics: ite$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$ *is true iff*
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \vee (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

Operational semantics: ite$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$
rewrites to the following guarded-constraints:
$c \longrightarrow C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$\neg c \longrightarrow C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \longrightarrow \neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}$
$\neg(\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1}) \longrightarrow c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}$
$(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_0}) \uplus (\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_1})$

The former two guarded–constraints result from the operational semantics of the **if_else** statement whereas the three latter allow more effective deductions. Particularly, the last constraint contains the constructive disjunction operator $\uplus$. This operator joins the results computed in both branches of the conditional. For example, if ite$(C, \begin{bmatrix} X_0 \end{bmatrix}, \begin{bmatrix} X_1 \end{bmatrix}, \begin{bmatrix} X_2 \end{bmatrix}, X_0 = 1, X_1 = 3)$ holds then the constructive disjunction operator leads to deduce that $X_2 \in \{1, 3\}$. Note that the `ite/6` combinator is awaked by the solver when the domain of at least one of its variable has changed. For example, learning that $X_2 \geq 2$ prunes the domain of X_2 to $\{3\}$, awakes the combinator and triggers its third guarded–constraint, as the constraint $X_2 \neq X_0$ is entailed by $dom(X_0) \cap dom(X_2) = \emptyset$. Hence, the constraint $\neg C$ is added to the constraint store and the `ite` combinator is removed from it.

Iterative statement. The SSA **while** statement
$\vec{v_2} = \phi(\vec{v_0}, \vec{v_1})$ **while** (*exp*) **{** *stmt* **}** is treated with the recursive user-defined combinator w$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$. When evaluating `w/5`, it is necessary to allow the generation of new constraints and new variables with the help of a substitution mechanism. `w/5` is defined as[3]:

Definition 2 `w/5`

Declarative semantics: w$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$ *is true iff*
$(c \wedge C_{Body} \wedge$ w$(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body})) \vee (\neg c \wedge \vec{v_2} = \vec{v_0})$

Operational semantics: w$(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$ *rewrites to*
$c \longrightarrow (C_{Body} \wedge$ w$(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$\neg c \longrightarrow \vec{v_2} = \vec{v_0}$
$\neg(c \wedge C_{Body}) \longrightarrow (\neg c \wedge \vec{v_2} = \vec{v_0})$
$\neg(\neg c \wedge \vec{v_0} = \vec{v_2}) \longrightarrow (c \wedge C_{Body} \wedge$ w$(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$(c \wedge C_{Body} \wedge$ w$(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body})) \uplus (\neg c \wedge \vec{v_2} = \vec{v_0})$

Note that the vector $\vec{v_3}$ is a vector of fresh variables. The first two guarded–constraints come from the operational semantics of a while in an imperative language. The last two come from the following observations: first, if the constraints extracted from the body are proved to be contradictory with the current constraint system then the loop cannot be entered; second, if any variable possesses distinct values before and after the execution of the **while** statement, then the loop must be entered at least once.

[3] For the sake of clarity, the constraint c generated through the substitution mechanism is not distinguished from c itself

3 An overview of the approach

Consider the task of generating a test datum on which branch 9-10 is executed in the C program of Fig.2. The process is composed of three main steps. The first step aims at generating the Pointer SSA form (PSSA), which is given in the second column of Fig.2. The definition of PSSA is mainly based on two ideas:

(1) to exploit the results of a specific pointer analysis, namely a points-to analysis, in order to perform all the hidden definitions. A points-to analysis is a static analysis that determines the set of memory locations that can be accessed through pointer dereferences. For every variable p of pointer type, a points-to analysis computes a set of variables that may be pointed by p during the execution. For example, at statement 7 of function foo, a *points-to analysis* says that p can (only) points to j or k. Note that the analysis usually overestimates the set of pointing relations that could exist during execution.

(2) to introduce two new forms of ϕ–functions to model the dereferencing process. A ϕ_u–function models the use of a dereferenced pointer: it returns one of its arguments depending on the points-to relations. For example, at statement 7,
$\phi_u(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$ returns j_3 if p points to j while it returns k_1 if p points to k.
ϕ_d–functions are used to reveal the hidden definitions realized through dereferenced pointers. At statement 8,
$\begin{bmatrix} j_4 \\ k_2 \end{bmatrix} = \phi_d(p_3, \begin{bmatrix} \&j \\ \&k \end{bmatrix}, r_1 * i, \begin{bmatrix} j_3 \\ k_1 \end{bmatrix})$, assigns $r_1 * i$ to j_4 if p points to j and j_3 otherwise, it assigns $r_1 * i$ to k_2 if p points to k and k_1 otherwise.

The second step of our approach translates the PSSA form into a CLP(FD) clause as shown in the third column of Fig.2. The clause head takes I and J_4 as arguments. I refers to the FD_variable generated for the input variable i, whereas J_4 refers to the variable that determines whether the branch 9-10 is executed or not. In this translation, each variable address is associated to a unique integer, noted $\&j$ for a SSA-variable j_i. ($\&j = 21$, $\&k = 22$ where 21 and 22 correspond to internal symbol table keys [4]) and specific CLP(FD) combinators extend ϕ_u and ϕ_d functions. These combinators maintain a **relation between their arguments**. So, partial information such as the variation domain of an argument, can be exploited to shrink the domain of the others.

[4] Keys from 0 to 20 are reserved to special symbols. For example, 0 represents the NULL pointer

Note that the ϕ_d combinator is related to the *IsAlias* function that was formerly introduced by Cytron and Gershbein [19] to realize hidden definitions in SSA form. Our approach distinguishes by providing relations and not only functions to model the use and definition of dereferenced pointers.

Finally, the last step consists in generating a request by making use of the control-dependencies of the program. Reaching branch 9-10 in the C code of the example requires $J_4 > 8$ hence the request shown in Fig.3 is generated. In

```
?- J4 > 8, foo(I, J4).
   I = 5 ;          /* first solution and forces backtracking */
   no               /* no other solution */
```

Figure 3. A test data generation request

this example, the result of the request says that there exists only a single test datum ($i = 5$) satisfying the request. If we examine the resolution process, we see that the three constraints $J_3 \in \{0, 2\}$ (deduced by the ite operator), $J_4 > 8$ and $\begin{bmatrix} J_4 \\ K_2 \end{bmatrix} = \Phi_d(P_3, \begin{bmatrix} 21 \\ 22 \end{bmatrix}, R_2, \begin{bmatrix} J_3 \\ K_1 \end{bmatrix})$ entails $P_3 = 21$ and $J_4 = R_2$ as $dom(J_3) \cap dom(J_4) = \emptyset$. As a consequence, $P_3 = P_2$ is refuted and the then–part of the conditional must be executed leading to $I < 6$. Finally, the constraints $R_1 = J_3$ and $R_2 = R_1 * I$ implies $I > 4$ which leads to $I = 5$ as the only solution.

The interesting point is that the combinator Φ_d provokes the assignment of the pointer variable P_3. In this example, numeric information over integer variables is used to refine pointer relationships.

4 The Pointer–SSA form (PSSA)

In this section, we introduce the PSSA form as an extension of SSA where pointer uses and definitions are treated with special functions that are defined with the help of a points-to analysis. Note that our proposition differs from other work where the goal is to exploit SSA in order to improve the accuracy of a points-to analysis [20], as our will is only to preserve the properties of SSA in the presence of conditional aliasing problems.

4.1 A simple language over pointer variables

In this paper, we will confine ourselves to a simple language over pointers: a structured language over multi-level pointers toward statically named variables (stack-directed pointers). In programs that use this class of pointers, the only operations that are allowed on pointers are (multiple) dereferencing (e.g. $**p$), addressing ($\&q$), pointer assignment ($p = q$), and pointer comparison ($p == q$, $p! = q$). Further, we suppose that programs are structured and do not contain unconstrained pointer arithmetic, type casting through pointers, pointers to functions or pointers to dynamically allocated structures. This paper is devoted to the treatment of pointers in the context of automated testing of C programs at the unit level, meaning that function calls are supposed to be stubbed or inlined.

4.2 Normalization

Normalizing a function consists in breaking complex statements into a set of elementary statements by introducing temporary variables. It is well-known that most C programs can be automatically translated in a program implemented with only a set of fifteen elementary statements [21,22,23]. In particular, a multi-level dereferenced pointer can be translated into a set of single dereferenced pointer by introducing temporary variables without modifying the program semantics. Fig. 4 contains a few examples of normalization that can easily be generalized to other statements. Note however that normalization is not required when a statement holds over non-pointer types (for example, $*p = *q$ does not need to be normalized if p and q are of pointer–to-integer type).

This normalization process allows to reason on a small number of statements without any loss of generality. Hence, the treatment of only four assignment statements are presented: $p = \&q$, $p = q$, $p = *q$, $*p = q$.

4.3 A Points–to analysis

As previously said, a *points–to analysis* statically collects a set of variables that may be pointed by the pointers of the program and determines the set of memory locations that can be accessed through a dereferenced pointer. In our work, we have chosen a points-to analysis previously introduced by *Emami et al.* [21]. In this analysis, a points-to relation is a triple: $pto(p, a, definite)$ or $pto(p, a, possible)$ where a denotes a variable pointed by p. In the former case, p points definitely to a on any control flow path that reaches the state-

Original code	Normalized code
$p = ***q$;	$tmp_1 = *q$; $tmp_2 = *tmp_1$; $p = *tmp_2$;
$**p = q$;	$tmp_1 = *p$; $*tmp_1 = q$;
$*p = \&q$;	$tmp_1 = \&q$; $*p = tmp_1$;
$*p = *q$;	$tmp_1 = *q$; $*p = tmp_1$;

Figure 4. Examples of normalization

ment where the pointing relation has been computed. In the latter case, p may points to a only on some control flow paths. However, the analysis does not say whether there exists a feasible control flow path that contains the pointing relation. Points-to relations can be captured by labeled directed graph (V, E), called the points-to graph. V denotes the set of vertices that corresponds to the variables of the program, while E denotes the set of labeled edges associated to the points-to relations. There exists an edge between p and a two variables iff $pto(p, a, definite)$ or $pto(p, a, possible)$ is true. Labels can be either symbolic (such as *possible* or $definite$) or expressions that denote the conditions under which the relation holds. This graph is computed on every statement of the program. Examples of points-to graphs are given in Section 6. Although it can be very inacurate, a points-to analysis is always conservative, meaning that if p points to a during an execution of the program then the results of the *points–to analysis* contains at least $pto(p, a, possible)$.

Points–to analysis can be either flow-sensitive and flow-insensitive, which is just a way to control the cost/accuracy tradeoff of the analysis. In the former case, the order on which the statements are executed is taken into account and the analysis is computed on each statement of the program. In the second case, the order is just ignored and the results of the points-to analysis are the same for all the statements. A flow-sensitive analysis is usually more accurate than a flow-insensitive but it is also more costly to compute. Fig.5 shows the difference between these two analyses on a very small piece of C code.

In our approach, we use a flow-sensitive analysis for the two following main reasons:

(1) when a statement contains a definition of a dereferenced pointer, every

C Code	Flow-sensitive on statement 3	Flow-insensitive
1. $p = \&a$;		pto(p,a,possible)
2. $q = p$;		pto(p,b,possible)
3. $p = \&b$;	pto(p,b,definite)	pto(q,a,possible)
	pto(q,a,definite)	pto(q,b,possible)

Figure 5. *Points-to analysis*

pointing relation hides a possible definition, hence the accuracy of the analysis directly plays on the number of hidden definitions;

(2) efficient algorithms exist for structured C functions.

Our method makes use of the syntax-based algorithm given in [21] to compute a flow-sensitive *points–to analysis*. In this algorithm, every statement can modify the pointing relations by maintaining the set of "killed" relations ($kill\_set$) and the set of relations generated by the statement (gen_set). The notation for both sets makes use of existentially quantified variables, denoted by "$\_x$" in this paper. For example, $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$ denotes the set of all pointing relations associated with p in the set In. Fig.6 presents the algorithm for elementary statements. For control flow structures, the results of the analysis on every branch are merged in a single set. In this merge process, a definite *points–to* relation can be transformed into a possible one. For loop statements, a fixpoint is computed by iterating on the body of the loop until no more modification can be exercised. Fig.7 shows the algorithm that handle the basic control flow structures. As the total number of possible points-to relations is bounded in the program (there is no dynamic allocation) and the merge process can only increase the set of points-to relation, existence and unicity of the fixpoint can easily be shown.

4.4 ϕ_u– and ϕ_d– functions in PSSA

In PSSA, ϕ_u–function models the use of a dereferenced pointer. Let $\begin{bmatrix} a_1 \\ .. \\ a_n \end{bmatrix}$ and $\begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix}$ denote two vectors of n program's variables, let $\begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}$ denotes the vector of distinct addresses of the first vector and p be a pointer variable, then the

// Given statement S and In a set of pointing relations
// process_basic(S, In) returns the set of
// pointing relations after S

Points–to process_basic(**Statement** S, **Points–to** In)

Case of

S is of the form $p = \&q$ **then**
$kill\_set := \{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
$gen\_set := \{pto(p, q, definite)\}$;

S is of the form $p = q$ **then**
$kill\_set := \{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
$gen\_set := \{pto(p, \_a, \_rel) | pto(q, \_a, \_rel) \in In\}$;

S is of the form $p = *q$ **then**
kill_set := $\{pto(p, \_x, \_rel) | pto(p, \_x, \_rel) \in In\}$;
gen_set := $\{pto(p, \_b, \_rel) | pto(q, \_a, \_r_1)$ and $pto(\_a, \_b, \_r_2) \in In\}$;
If $\_r_1$ and $\_r_2$ are $definite$ **then** $\_rel = definite$ **else** $\_rel = possible$

S is of the form $*p = q$ **then**
kill_set := $\{pto(\_x, \_y, \_rel) | pto(p, \_x, definite)$ and $pto(\_x, \_y, \_rel) \in In\}$;
gen_set := $\{pto(\_x, \_z, \_rel) | pto(p, \_x, \_r_1)$ and $pto(q, \_z, \_r_2) \in In\}$;
If $\_r_1$ and $\_r_2$ are $definite$ **then** $\_rel = definite$ **else** $\_rel = possible$

returns $(Input \setminus kill\_set) \cup gen\_set$;

Figure 6. Flow–sensitive points–to analysis of basic statements

ϕ_u–function $\phi_u(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$ returns v_i if $p = \&a_i$. Note that each $\&a_i$ is a distinct constant. In PSSA, ϕ_d–function models the definition of a dereferenced pointer. A ϕ_d–function $\phi_d(p, \begin{bmatrix} \&a_1 \\ .. \\ \&a_n \end{bmatrix}, expr, \begin{bmatrix} v_1 \\ .. \\ v_n \end{bmatrix})$, returns a vector of variables

```
/* Given statement S and In a set of pointing relations */
/* process(S, In) returns the set of relations after S*/

Points–to process( Statement S, Points–to In)

    If S is void then returns In

    If S is a list of basic statements then
        Head := pop(S) /* returns the head of S */
        Tail := tail(S)  /* returns the tail of S */
        Out := process_basic(Head, In) ;
        returns process(Tail, Out) ;

    If S is of the form [if(C) then S1 else S2]
    then
        Out_then := process(S1,In) ;
        Out_else := process(S2,In) ;
        returns merge(Out_then, Out_else) ;

    If S is of the form [while(C) do S]
    then
        do
            LastIn := In ;
            Out := process(S, In) ;
            In := merge(In, Out) ;
        while LastIn ≠ In
        returns In
```

Figure 7. Flow-sensitive points–to analysis of control structures

$\begin{bmatrix} x_1 \\ .. \\ x_n \end{bmatrix}$ where $x_i = expr$ if $p = \&a_i$ and $x_j = v_j$ for all $j \neq i$.

4.5 Building the PSSA form

A few notations are required to describe the algorithm used to build the PSSA form. When analyzing a statement, $p\downarrow$ denotes the last numbered variable associated to p whereas $p\uparrow$ denotes the new fresh numbered variable for p. So, a definition of p is noted $p\uparrow$ and a use of p is noted $p\downarrow$. If PTO_S denotes the set of pointing relations available at statement S, then $from(p, PTO_S)$ is the set of variables pointed by p. Formally speaking, $from(p, PTO_S) = \{\_a \mid pto(p, \_a, \_) \in PTO_S\}$.

The algorithm that builds the PSSA form works on each statement by generating ϕ_u or ϕ_d functions each time a dereferenced pointer is encountered. Fig.8 contains the algorithm for basic statements obtained after normalization, bearing in mind that other statements can easily be deduced from the treatment of these ones.

```
/* Given statement S, PTO_S a set of pointing relations*/
/* pointer_ssa(S, Input) returns a statement*/
/* to add to the PSSA form */

Statements pointer_ssa(Statement S, Points–to PTO_S)

  Case of
    S is of the form p = *q then
      q⃗ := vector_of_addresses(from(q, PTO_S)) ;
              /* addresses of aliased variables of (*q) */
      v⃗ := vector_of_dereferenced(from(q, PTO_S));
              /* dereferenced aliased variables of (*q) */
      St := (p↑= φ_u(q↓, q⃗, v⃗↓));

    S is of the form *p = q then
      p⃗ := vector_of_addresses(from(p, PTO_S)) ;
              /* addresses of aliased variables of (*p) */
      v⃗ := vector_of_dereferenced(from(p, PTO_S));
              /* dereferenced aliased variables of (*p) */
              St := (v⃗↑= φ_d(p↓, p⃗, q↓, v⃗↓))

    S is of any other form then
      St := SSA(S) ; /* Standard SSA */
  return St ;
```

Figure 8. Algorithm for constructing PSSA

5 Combinators Φ_u and Φ_d in CLP(FD)

As a result of the PSSA translation, the operators '&' and '*' of the C language have been removed without any loss of semantics. In PSSA, two new functions have been introduced: ϕ_u– and ϕ_d– functions. These functions are translated into two new relational combinators in the CLP(FD) program. The definition of these CLP(FD) combinators is based on guarded–constraints as done for both `ite/6` and `w/5`. The Φ_u combinator maintains a relation between a pointer, the set of possibly pointed variables and a variable to be assigned.

It exploits the fact that, during an execution, a pointer can only point to a single variable.

Definition 3 $\Phi\_u/4$

Declarative semantics : *Let $X, P, V_1, .., V_n$ be FD_variables and let $P_1, .., P_n$ be n distinct numeric constants, then $X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ is true iff $\exists i | P = P_i \wedge X = V_i$.*

Operational semantics : $X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$

rewrites to:
$dom(X) := dom(X) \cap (\cup_{i=1}^{n} dom(V_i))$,

if $n = 0$ then **fail** *else forall i in $1..n$ do*

$(P = P_i) \longrightarrow X = V_i$,
$\neg(X = V_i \wedge P = P_i) \longrightarrow P \neq P_i \wedge$

$$X = \Phi_u(P, \begin{bmatrix} P_1 \\ .. \\ P_{i-1} \\ P_{i+1} \\ .. \\ P_n \end{bmatrix}, \begin{bmatrix} V_1 \\ .. \\ V_{i-1} \\ V_{i+1} \\ .. \\ V_n \end{bmatrix}),$$

The Φ_d combinator maintains a relation between a pointer, a variable associated to the dereferenced pointer, the set of possibly pointed variables, and the set of possibly assigned variables.

Definition 4 $\Phi\_d/4$

Declarative semantics : *Let $X, P, V_1, .., V_n$ be FD_variables and $P_1, .., P_n$ be n be distinct numeric constants, then $\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ is true iff $\exists i | P = P_i \wedge X_i = DP \wedge \{X_j = V_j\}_{\forall j \neq i}$.*

Operational semantics: $\begin{bmatrix} X_1 \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_n \end{bmatrix})$ *rewrites to:*

$dom(DP) := dom(DP) \cap (\cup_{i=1}^{n} dom(X_i))$,

if $n = 0$ then **fail** *else forall i in $1..n$ do*

$dom(X_i) := dom(X_i) \cap (dom(DP) \cup dom(V_i))$
$\neg(X_i = V_i \wedge P \neq P_i) \longrightarrow (P = P_i \wedge X_i = DP \wedge \{X_j = V_j\}_{j \neq i})$,

$\neg(X_i = DP \wedge P = P_i) \longrightarrow P \neq P_i \wedge X_i = V_i$

$$\wedge \begin{bmatrix} X_1 \\ .. \\ X_{i-1} \\ X_{i+1} \\ .. \\ X_n \end{bmatrix} = \Phi_d(P, \begin{bmatrix} P_1 \\ .. \\ P_{i-1} \\ P_{i+1} \\ .. \\ P_n \end{bmatrix}, DP, \begin{bmatrix} V_1 \\ .. \\ V_{i-1} \\ V_{i+1} \\ .. \\ V_n \end{bmatrix}),$$

When p is assigned to an invalid address, then both Φ_d and Φ_u combinators **fail** during the solving process. As a failure in CLP corresponds to the unsatisfiability of the constraint store, this correctly entails that the current subpath under investigation is non-feasible.

6 Preliminary results

6.1 The InKa tool

We implemented our approach within the test data generator INKA [2,14]. The tool automatically generates test data for the coverage of structural criteria such as all_statements and all_decisions. It handles a non-trivial subset of the C and C++ languages [13]. The tool has several other functionalities, such as test

coverage measurements, control flow monitoring and test cases management. It is mainly developed in Prolog, Java and C and makes use of the clp(fd) library of Sicstus Prolog [18] to solve the test data generation requests. Our current implementation includes a pointer analyzer, a PSSA form generator and the design of both combinators Φ_u and Φ_d.

	Normalized C Code	PSSA form
	int lh98(**int** h)	**int** lh98(**int** h_0)
	int $g, **p, *q, *r$;	**int** $g, **p, *q, *r$;
1.	$g = 3$;	$g_1 = 3$;
2.	$q = \&h$;	$q_1 = \&h$;
3.	$r = \&g$;	$r_1 = \&g$;
4.	$p = \&r$;	$p_1 = \&r$;
5.	**if**($h < 10$)	**if**($h_0 < 10$)
6.	$g = (h + 2) * 5$;	$g_2 = (h_0 + 2) * 5$
7.	$p = \&q$;	$p_2 = \&q$;
		$\begin{bmatrix} g_3 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} g_2 \\ p_2 \end{bmatrix}, \begin{bmatrix} g_1 \\ p_1 \end{bmatrix})$;
8.	$t = *p$;	$t_1 = \phi_u(p_3, \begin{bmatrix} \&q \\ \&r \end{bmatrix}, \begin{bmatrix} q_1 \\ r_1 \end{bmatrix})$;
		$tmp = \phi_u(t_1, \begin{bmatrix} \&h \\ \&g \end{bmatrix}, \begin{bmatrix} h_0 \\ g_3 \end{bmatrix})$;
9.	$h = 2 * g + *t$;	$h_1 = 2 * g_3 + tmp$;
10.	if($h > 100$) ;	if($h_1 > 100$) ;
11.	...	...

Figure 9. Pointer–SSA form of lh98

6.2 Experimental evaluation

To evaluate the approach, we generated test data for C functions that contain conditional pointer aliasing problems. In this paper, we report the experimental results on several programs extracted from the literature. Two of them are detailed as they introduce particular technical difficulties for goal-oriented test data generation. The first program, extracted from [22], is shown in Fig.9 along with its PSSA form. It contains an aliasing problem with two-level indirection pointers when one wants to reach branch 10-11. The points-to graph computed for program lh98 at statement 9 by the flow-sensitive points-to analysis is given by the following diagram:

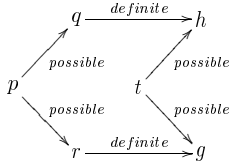


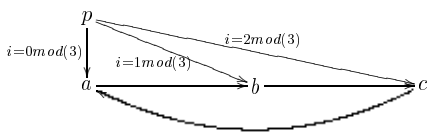
Fig.10 contains the CLP(FD) program generated for function lh98 and the requests asking for test data able to reach branch 10-11. The results show that


```
lh98(H0, H1, ...):-
    H0 ∈ −2^31..2^31 − 1,
    G1 = 3,
    Q1 = 21,
    R1 = 22,
    P1 = 23,
    ite(H0 < 10, [G1, P1], [G2, P2], [G3, P3], G2 = (H0 + 2) * 5 ∧ P2 = 24)
    T1 = Φu(P3, [24, 23], [Q1, R1]),
    TMP = Φu(T1, [21, 22], [H0, G3])
    H1 = 2 * G3 + TMP,
    ite(H1 > 100, ...).

?- H1 > 100, lh98(H0, H1, ...).

    H0 ∈ 8..9

?- H1 > 100, lh98(H0, H1, ...), labeling([H0]).

    H0 = 8 ;

    H0 = 9 ;

    no
```

Figure 10. CLP(FD) program for lh98

there are only two values for H_0 able to reach branch 10-11. In this example, the propagation step is so efficient that all the inconsistent values are removed from the domain of H_0, as shown by both requests. The first one gives the results of the propagation step: if there exists a value for H_0 that satisfies the request, then it belongs to 8..9. The second one makes choices and shows that both values (8 and 9) are actual solutions.

In the second program given in Fig.11, pointing relationships are modified within a loop which makes it difficult to analyze. At each iteration, p is assigned the value of $*p$ that has been computed by the previous iteration. As the points-to relations are cyclic in this example (a points-to b, b points-to c and c points-to a), the loop iterates over the possible points-to relations. The points-to graph computed by the flow-sensitive points-to analysis at statement 8 is as follows:



In this diagram, the edges are labeled by the conditions that hold over i in order to satisfy a given points-to relation. For example, p points-to c iff $i = 2mod(3)$. Hence, statement 9 ($p == \&c$ is satisfied) is executed all three loop

	Normalized C code	PSSA form
	int jos97(**int** i)	**int** jos97(**int** i_0)
	int $****p, ***a, **b, *c$;	**int** $****p, ***a, **b, *c$;
1.	$p = \&a$;	$p_1 = \&a$;
2.	$a = \&b$;	$a_1 = \&b$;
3.	$b = \&c$;	$b_1 = \&c$;
4.	$c = \&a$;	$c_1 = \&a$;
		$\begin{bmatrix} i_2 \\ p_3 \end{bmatrix} = \phi(\begin{bmatrix} i_0 \\ p_1 \end{bmatrix}, \begin{bmatrix} i_1 \\ p_2 \end{bmatrix})$;
5.	**while**($i > 0$)	**while**($i_2 > 0$)
	do	**do**
6.	$i = i - 1$;	$i_1 = i_2 - 1$;
7.	$p = *p$;	$p_2 = \phi_u(p_3, \begin{bmatrix} \&a \\ \&b \\ \&c \end{bmatrix}, \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix})$;
8.	**if**($p == \&c$)	**if**($p_3 == \&c$)
9.	**return** 1 ;	**return** 1 ;
	else	**else**
10.	**return** 0 ;	**return** 0 ;

Figure 11. PSSA form of function jos97

iterations, starting from the second one. These conditions were determined manually in order to check our implementation.

Fig.12 contains the CLP(FD) program generated for jos97. The first request asks for a test datum to be generated to reach branch 8-9. It uses only constraint propagation. The second request includes a labeling call. With the first request, the solver prunes the domain of I_0 to $2..2^{31} - 1$, which is, as expected, a correct over-estimation of the variation domain of I_0. Note however that not all the values of this domain are solutions of the problem. The low bound of the interval is 2, showing that two non-feasible paths where automatically detected (paths 1-2-3-4-5-8-9 and 1-2-3-4-5-6-7-5-8-9) by the constraint propagation process. The second request gives the correct solutions through the backtracking process of Prolog. The values found by the solver correspond to the values that satisfy the constraint $i = 2mod(3)$. These two requests show that the pointing relations were correctly propagated by the solving process.

In addition to our results for foo (Fig.2), lh98 (Fig.9) and jos97 (Fig.11), test data were generated for several other programs: ow91 extracted from [24], lm03 from [25], bl98 from [26], a version of the famous trityp program [5] that contains conditional pointer aliasing problems and a version of the jos97 program in which the number of points-to relation appears as a parameter and thus, can be easily increased. For each program, we generated a test set that guarantees the complete coverage of the *all_decisions* criterion with InKa. This criterion requires that every decision is covered at least once during the testing phase. We compared our approach with a random test data generation technique which blindly generates test data until a given criterion is covered.

jos97(I_0, P_3, R_3) :-
$I_0 \in -2^{31}..2^{31} - 1$,
$P_1 = 21$,
$A_1 = 22$,
$B_1 = 23$,
$C_1 = 21$,
`while`(
$I_2 > 0, \begin{bmatrix} I_0 \\ P_1 \end{bmatrix}, \begin{bmatrix} I_1 \\ P_2 \end{bmatrix}, \begin{bmatrix} I_2 \\ P_3 \end{bmatrix},$
$I_1 = I_2 - 1 \wedge$
$P_2 = \Phi_u(P_3, \begin{bmatrix} 21 \\ 22 \\ 23 \end{bmatrix}, \begin{bmatrix} A_1 \\ B_1 \\ C_1 \end{bmatrix})$
),
`ite`($P_3 = 23, \begin{bmatrix} R_1 \end{bmatrix}, \begin{bmatrix} R_2 \end{bmatrix}, \begin{bmatrix} R_3 \end{bmatrix}, R_1 = 1, R_2 = 0$).

?- $P_3 = 23$, jos97(I_0, P_3, R).

$I_0 \in 2..2^{31} - 1$
$R = 1$

?- $P_3 = 23$, jos97(I_0, P_3, R), `labeling`($[I_0]$).

$I_0 = 2$ /* first solution */
$R = 1$;

$I_0 = 5$ /* second solution */
$R = 1$;

$I_0 = 8$ /* third solution */
$R = 1$;

$I_0 = 11$ /* fourth solution */
$R = 1$;

... /* all the solutions */

Figure 12. CLP(FD) program for jos97

As random testing can be developed very easily, we argue that every new automated test data generation technique should be shown to outperform random testing. Note however that random testing is unable to show the unfeasibility of a decision, unlike our approach. For random testing, in order to provide a fair comparison with a randomized technique, we also evaluated the *almost_all_decisions* criterion which only requires all but one decisions being covered.

The results given in Tab.1 were computed on a 1.8Ghz Pentium 4 personal computer with 512Mb of RAM. In the left side of the table, we report the number of control flow paths (#p), the number of unfeasible paths (#u) deduced by a manual analysis, and the maximum number of edges in the points-to graphs of the program (#a). These parameters give an insight into the difficulty of conditional pointer aliasing problems in the tested programs. Typically, a path-oriented method would probably fail on a program that contains

a lot of non-feasible paths. We also report two statistics automatically computed by the tool: the number of constraints that are dynamically added to the constraint store (#c) and the number of times a domain is reduced (#pr). These data characterize respectively the size of the constrained problem and the pruning capacity of constraints. We report the results computed by InKa and by the random testing approach. To minimize the development effort and to provide a fair comparison, we used the constrained program model itself to implement random testing. Random numbers were generated inside our tool and tested again the constraint system generated for each program by following a generate–and–test approach. In the random approach, the variation domains were shrinked to -1000 .. 1000 and the time allocated to the evaluation of a single test data was restricted to 4sec of CPU time to avoid long-term computations. In Tab.1, #td corresponds to the number of test data generated for the complete coverage of a criterion, and rt corresponds to the CPU time required to do so. Note that our test data generation strategy is not optimal and so does not lead necessarily to the minimum number of test data required for the full coverage of the criterion. For random testing, we repeated ten times the experience to avoid the factor of bad "luck" introduced by the starting point sequence of random numbers. The values that are shown in the latter columns of Tab. 1 correspond to the mean values of test data generated and runtime measured for the coverage of the *all_decisions* and the *almost_all_decisions* criteria. When testing objectives were not fulfilled within 1 hour of user time, we stopped the random generation and reported failure (shown with "-").

Table 1
Criteria coverage in the presence of conditional aliasing pointers

	program features			Our approach				Random approach			
				all_decisions				all_decisions		almost_all_decisions	
Programs	#p	#u	#a	#c	#pr	#td	rt(sec)	#td	rt(sec)	#td	rt(sec)
`ow91`	3	0	3	34	259	3	0.5	-	-	1417	2.5
`lm03`	4	2	1	5	27	2	0.1	3278	3.07	2215	1.9
`lh98`	4	0	6	36	174	2	0.1	1303	1.93	3	0.1
`bl98`	4	0	7	47	1042	3	0.1	3199	4.45	1007	1.4
`foo`	8	6	2	100	680	2	0.1	-	-	-	-
`trityp`	57	43	4	30734	80822	8	5.3	-	-	2870	12.1
`jos97`	∞	∞	6	106796	117802	2	18.0	5	1.27	2	0.4

6.3 Analysis

In all the cases, our approach achieves the full coverage of the *all_decisions*, even when the maximum number of points-to relations is high (bl98) or the number of infeasible paths is high (trityp) or infinite (jos97). By looking at the number of prunings, we confirm our intuition that the constraints play

an active role in the generation. Let us recall that this number corresponds to the number of times a domain is pruned by a constraint. The program jos97 is probably the more demanding as shown by the number of constraints generated and the number of prunings performed and requires so the greatest CPU runtime value (18sec). In all other cases, our approach outperforms random testing in terms of CPU time required to get the coverage of a criterion. The number of test data generated is not significative as our approach is deterministic while the random testing approach is probabilistic. In some cases (ow91, foo and trityp), the random testing approach fails to get a test set that covers the selected criterion. This is due to the fact that several decisions have a very low probability to be satisfied, especially in the presence of conditional aliasing problems. In our approach, the search of solutions is "guided" by the constraint solving process, even in the presence of pointers. We tried to change several parameters in random testing (such as the size of domains, or the CPU time allocated to the evaluation of a single test data) to get better results . However, this does not change the quality of results as exemplified by the following evidence. In the trityp program, there is a decision that corresponds to the event $i = j = k$ where i, j, k represents the lengths of a triangle. It is trivial to see that the event which consists to generate a triple of equal numbers has a very low probability to happen whatever be the modified parameters. The program `jos97` distinguishes by the fact that all its decisions can be covered very quickly by the random testing approach. Only 5 test data in average are required to do so and the coverage can be obtained in 1.27sec in average. On the contrary, our approach requires 18sec to set up and solve the constraint system generated for this program. This is due to the fact that every time the loop of the program is unrolled, the solving process examines all the 6 possible points-to relations.

7 Related work

To the best of our knowledge, prior work on automatic goal-oriented test data generation did not address specifically pointer aliasing problems. However, several test data generation approaches deal with pointer variables. Korel [27] proposed exploiting several executions of the program to find a test datum on which a selected path is executed. Its approach does not suffer from the pointer aliasing problem as it is based solely on program executions. As a drawback, this approach cannot detect unsatisfiability of selected paths. More recently, Visvanathan and Gupta in [28], Marre et al. in [29] and Sai-ngern et al. [30] addressed the problem of generating test data for C functions with pointers as input parameters by using symbolic execution and constraint solving techniques. In their approaches, pointer relationships are handled with constraints on input values and aliasing problems occur only within input data structures.

PathCrawler [29] and CUTE [31] are two similar test data generators that try to cover all the feasible paths of C programs. Both systems work by combining concrete and symbolic execution. They solve path conditions in order to find the next test data that will follow a path that improve the current coverage of the program. All these approaches have in common the need for a path to be selected first and so fall in the path-oriented methods category. Unlike path-oriented and among other advantages, goal-oriented methods exploit the early detection of non-feasible paths to prune the search space made up of all the paths that reach a given branch [32]. However, unlike path-oriented ones, goal-oriented methods suffer from the conditional pointer aliasing problem. Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths can be exponential on the number of decisions of the program or even infinite when loops are unbounded. As a consequence, we argue that goal-oriented methods scale up more easily than path-oriented methods for programs that contain only pointer variables. The algorithmic complexity of our approach is strongly related to the maximum number of possible aliases computed at each point of the program and there are studies showing this number remain small in general. For example, the experimental results section of [21] shows this number is almost always less than five. For us, the main drawback of our approach concerns its weakness to deal with dynamic allocation. Indeed, allowing dynamic allocation into a while-loop leads to the creation of a potentially unbounded number of aliases. Hence, the points-to analysis we used is no more suitable in this case. Note however that dealing with dynamic allocated structures in a goal-oriented method remains an open problem and none of the work previously cited addressed it.

8 Conclusion

In this paper, we have presented a new goal-oriented method for generating automatically test data for programs with multi-level pointer variables. The method is based 1) on the Pointer SSA form that extends traditional SSA by integrating the results of an intraprocedural flow-sensitive pointer analysis and 2) on the design of two CLP combinators that model the relation between pointers and pointed variables. The next steps of this work will be to study extensions in several directions. First, our approach could address the problem of function calls by exploiting the results of an interprocedural pointer analysis. Although a lot of work has been done in this area, technical problems remain to handle properly function pointers (second-order programming) and recursive calls. Second, we would like to extend our approach for pointers that address the heap. We could use other pointer analysis such as a shape analysis or dynamic points-to analysis to deal with dynamic allocation. These two extensions could open the road to the development of a symbolic goal-oriented

test data generation method able to deal with real-sized programs.

Acknowledgments

Thanks to Nicky Williams for her helpful comments on an earlier draft of this paper.

References

[1] A. Gotlieb, B. Botella, M. Rueher, Automatic test data generation using constraint solving techniques, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, FL, USA, 1998, pp. 53–62.

[2] A. Gotlieb, B. Botella, M. Rueher, A clp framework for computing structural test data, in: Proceedings of Computational Logic (CL'2000), LNAI 1891, London, UK, 2000, pp. 399–413.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. Zadeck, Efficently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Language and Systems 13 (4) (1991) 451–490.

[4] S. Kowshik, D. Dhurjati, V. Adve, Ensuring code safety without runtime checks for real-time control systems, in: Proc. of the Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES'02), Grenoble, FR, 2002.

[5] R. DeMillo, J. Offut, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.

[6] R. DeMillo, J. Offut, Experimental results from an automatic test case generator, ACM Transactions on Software Engineering Methodology 2 (2) (1993) 109–127.

[7] J. Offut, Z. Jin, P. J., The dynamic domain reduction procedure for test data generation, Software–Practice and Experience 29 (2) (1999) 167–193.

[8] C. Meudec, ATGen: automatic test data generation using constraint logic programming and symbolic execution, Software Testing, Verification and Reliability 11 (2) (2001) 81–96.

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

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

[11] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Language and Systems 9 (1987) 319–349.

[12] M. Brandis, H. Mőssenbőck, Single-Pass Generation of Static Single-Assignment Form for Structured Languages, ACM Transactions on Programming Language and Systems 16 (6) (1994) 1684–1698.

[13] Axlog Ingenierie and Thales Airborne Systems, INKA–V1 User's Manual (december 2002).

[14] A. Gotlieb, B. Botella, Automated metamorphic testing, in: 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC'03), Dallas, TX, USA, 2003.

[15] T. Chen, T. Tse, Z. Zhou, Fault-based testing in the absence of an oracle, in: IEEE Int. Comp. Soft. and App. Conf. (COMPSAC), 2001, pp. 172–178.

[16] K. Marriott, P. Stuckey, Programming with Constraints : An Introduction, The MIT Press, 1998.

[17] N. T. Sy, Y. Deville, Consistency techniques for interprocedural test data generation, in: ESEC/FSE-11: Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, 2003, pp. 108–117.

[18] M. Carlsson, G. Ottosson, B. Carlson, An open–ended finite domain constraint solver, in: Proc. of Programming Languages: Implementations, Logics, and Programs, 1997.

[19] R. Cytron, R. Gershbein, Efficient accommodation of may–alias information in SSA form, in: Proceedings of Programming Languages Design and Implementation, ACM, Albuquerque, NM, 1993.

[20] R. Hasti, S. Horwitz, Using static single assignment form to improve flow-insensitive pointer analysis, in: PLDI '98: Proc. of conference on Programming language design and implementation, ACM Press, 1998, pp. 97–105.

[21] E. Emami, R. Ghiya, L. Hendren, Context–sensitive interprocedural points–to analysis in the presence of function pointers, in: Proceedings of Programming Languages Design and Implementation, ACM, Orlando, FL, 1994.

[22] C. Lapkowski, L. Hendren, Extended SSA numbering: Introducing SSA properties to languages with multi–level pointers, in: 7th Proc. of the Conference on Compilers Construction (CC'98), LNCS 1383 Kai Koshimies (Ed), Lisbon, Portugal, 1998, pp. 128–143.

[23] R. Ghiya, L. Hendren, Putting pointer analysis to work, in: Proceedings of Symp. on Principles of Programming Languages, ACM, San Diego, CA, 1998.

[24] T. Ostrand, E. Weyuker, Data flow-based test adequacy analysis for languages with pointers, in: In Proceedings of the Symposium on Testing, Analysis, and Verification (TAV'91), 1991, pp. 74–86.

[25] V. Livshits, M. Lam, Tracking pointers with path and context sensitivity for bug detection in C programs, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03), 2003, pp. 317–326.

[26] D. Binkley, J. Lyle, Application of the pointer state subgraph to static program slicing, Journal of Systems and Software 40 (1) (1998) 17–27.

[27] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.

[28] S. Visvanathan, N. Gupta, Generating test data for functions with pointer inputs, in: Proceedings of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE'02), Edinburgh, UK, 2002.

[29] B. Marre, P. Mouy, N. Williams, On-the-fly generation of k-path tests for c functions, in: Proceedings of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04), Linz, Austria, 2004.

[30] S. Sai-ngern, C. Lursinap, P. Sophatsathit, An address mapping approach for test data generation of dynamic linked structures, Information and Software Technology 47 (2005) 199–214.

[31] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, in: 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05), ACM, 2005, pp. 263–272.

[32] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering Methodology 5 (1) (1996) 63–86.

F. Charreteur, B. Botella, and A. Gotlieb. *Modelling dynamic memory management in constraint-based testing*. **The Journal of Systems and Software, 82(11):1755–1766, Nov. 2009. Special Issue: TAIC-PART 2007 and MUTATION 2007.**

Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur[1]
University of Rennes 1
Campus de Beaulieu
35042 RENNES – France
Florence.Charreteur@irisa.fr

Bernard Botella
CEA LIST
Saclay
91 191 Gif SurYvette – France
Bernard.Botella@cea.fr

Arnaud Gotlieb
INRIA
Campus de Beaulieu
35042 RENNES – France
Arnaud.Gotlieb@irisa.fr

Abstract

Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques. When programs contain dynamic memory allocation and loops, constraint reasoning becomes challenging as new variables and new constraints should be created during the test data generation process. In this paper, we address this problem by proposing a new constraint model of C programs based on operators that model dynamic memory management. These operators apply powerful deduction rules on abstract states of the memory enhancing so the constraint reasoning process. This allows to automatically generate test data respecting complex coverage objectives. We illustrate our approach on a well-known difficult example program that contains dynamic memory allocation/deallocation, structures and loops. We describe our implementation and provide preliminary experimental results on this example that show the highly deductive potential of the approach.

Keywords: *Software Testing*, *Constraint-Based Testing, Automatic Test Data Generation, Dynamic structures*

1. Introduction

By increasing our confidence in the quality of software, testing techniques play a prevalent role in the verification process of software systems. However, software testing remains an expensive task in the development process and one of the main challenges concerns its possible automation. Since the seminal work of Offut and De Millo in the context of mutation testing [1], much attention has been devoted to the use of constraint solving techniques in the automation of software testing (*Constraint-Based Testing*). In [2], Dick and Faivre proposed to exploit a VDM model to generate automatically test cases for partition testing while Marre proposed in [3] to exploit Constraint Logic Programming to generate test cases from an algebraic specification. In 1998, Gotlieb, Botella and Rueher proposed using constraint propagation techniques to generate test data for the structural coverage of C programs [4]. This work resulted in the development of the INKA tool which was the first to address C programs containing pointers (but without dynamic allocation) and floating-point variables [5,6,7]. Meudec followed a similar path with ATGen, a software test data generator based on Constraint Logic Programming for ADA programs [8] and more recently, Williams introduced a new test data generation method in a tool called PathCrawler, based on the on-the-fly generation of paths in C programs [9]. This method was independently discovered by Godefroid and Sen in the DART and CUTE approaches [10,11]. Note that Constraint-Based Testing methods cover several applications area including hardware verification [12,13], test data generation for structural testing [4-11], functional testing [1,2], counter-example generation [14,15], and software verification [16].

Constraint-Based Testing (CBT) is a two-stage process consisting in generating test data against a user-selected testing objective. The first stage is a **constraint generation step** aiming at extracting a constraint system from the source code of a program under test and a selected testing objective, while the second stage is a **constraint solving step** consisting in trying to solve the constraint system in order to get a test data that satisfies the testing objective. For example, by selecting a statement within the program under test, we get a constraint system that characterizes a subdomain of the program input domain and solving this constraint system leads to generate a test data in the subdomain, on which the statement is executed. Such an approach has been called goal-oriented test data generation by Fergusson and Korel [17] as opposed to path-oriented test data generation which consists to select first a path within the program before trying to generate a test data that activates the corresponding path. When the constraint system has no solution, then the testing objective is unreacheable. For goal-oriented test data generation, it means that the corresponding statement or decision is unreacheable. Such deductions are usually outside of the scope of any path-oriented test data generators as soon as a loop is present in the program because the path selection process is possibly endless in this case. Note however that showing the unsatisfiability of a constraint system is undecideable in the general case[2] and then usually some unreacheable elements may remain undetected even for goal-oriented approaches.

This paper addresses the problem of dynamic memory management in a goal-oriented test data generation approach. Dynamic structures are heap-based data structures built during the execution of the program. CBT approaches cannot easily handle these structures, as their exact shape cannot be completely known at compile time. One usually resort either to use an approximation of the structure shape by static analyses or to use dynamic analysis. In the first case, one cannot make exact deductions about the states of the memory manipulated by the program as the program semantics has been approximated, while in the second case, deductions can only be performed on some program executions.

1.1 A motivating example

Let us illustrate this problem on a non-trivial example that belongs to the folklore of C pointer problems [18]. The Josephus program, shown in Fig. 1, is a decimation problem and relates to a Historical situation where Jewish rebels were surrounded by Romans and decided to commit to suicide: they lined up in a circle and systematically killed every other one, going around and around, until only one rebel is left. The program contains two successive loops: the first one builds a circular simple-linked list of n nodes, while the second eliminates nodes at position m until only a single node remains in the list. Thus, this program contains dynamic memory allocation/deallocation within loops, which is the main technical difficulty concerning dynamic structures in CBT approaches. Indeed, an interesting testing objective for the Josephus program is to find values for m and n such that the second loop (**while 2**) is unrolled at least forty times, as this number corresponds to the Ancient problem[3]. Such complex testing objectives are frequent in practice, as interesting states of a system often result from complex control flow. Note also that such states are usually difficult to reach by hand even for experienced validation engineers [19], making automation a real challenge. For CBT tools, finding values for m and n such that the second loop (**while 2**) is unrolled at least forty times

[1] Corresponding author – Florence Charreteur – Tel/Fax : +33 299 842 2 86/+33 299 847 171

[2] As a consequence of the 1970's Matiyasevitch result on the undecideability of the Tenth problem of Hilbert

[3] In the original ancient Josephus's decimation problem, there were 40 people killed, letting the rebel alive at position 31.

involves either complex static analyses such as structure sharing analysis and pointer aliasing analysis or dynamic analysis (based on program executions). A static analyzer would typically ignore the backward pointer toward the first element of the list, while a dynamic analyzer would have very few chances to satisfy our testing objective as they are usually based on initial random draws. On the contrary, thanks to our operators that model the dynamic memory management, our system automatically deduces that n, the length of the initial list should be 41 while the remaining element at the end of the process is 31 when m=3. This corresponds exactly to the expected solution of the original problem (the surviving rebel will be the one positioned at the 31th position in the circle).

```
typedef struct node *link;
struct node { int key ; link next;};

int f(int n,int m){
1. int i; link t,x ;
2. t=(link)malloc(sizeof(struct node));
3. t->key = 1;
4. x = t;
5. i = 2;
6. while( i <= n ){         //while1
7.   t->next=
      (link)malloc(sizeof(struct node));
8.   t = t->next ;
9.   t->key = i;
10.  i++;}
11.t->next = x ;
12.while( t != t->next){  //while 2
13.  i = 1;
14.  while( i <= m-1){   //while 3
15.   t = t->next ;
16.   i++;}
17.  x = t->next ;
18.  t->next = (t->next)->next ;
19.  free(x);}
```

Figure 1. Josephus program

1.2 Contributions

The main contribution of the paper is the design of several constraint operators that model memory allocation, deallocation, accesses and updates. These operators are equipped with deduction rules useful to perform constraint reasoning on imperative programs, as required by CBT tools. We present each operator under the form of finite state machine that interacts with the constraint solver. Such a presentation is advantageous as it makes clearer the implementation of these operators. Another contribution of the paper is the design of a complete goal-oriented test data generation method for C programs containing dynamically allocated structure. Our approach can deal with circular lists as well as any memory shapes that may include backward pointers. We are not aware of any other symbolic approaches able to deal with these data structures (see section 7). We implemented our constraint operators within the test data generator INKA [7] and got preliminary experimental results that show the potential of the constraint operators to reason about program with dynamic memory management.

1.3 Outline of the paper

Section 2 recalls the necessary background on constraint solving over finite domains and states some notations and restrictions. Section 3 introduces our overall goal-oriented constraint-based test data generation technique. Section 4 details the abstract memory model we use to deal with dynamically allocated structures. Section 5 presents the deduction rules exploited in the constraint operators under the form of abstract state machines; Section 6 gives our preliminary experimental results while section 7 discusses related works. Finally, Section 8 concludes and draws some perspectives to this work.

2. Background

2.1 Constraint solving over finite domains

Our approach is based on constraint solving over finite domains. In this framework, a finite domain is associated to each variable and a solution to the constraint system is a valuation of the variables within their domains that satisfy each constraint. Primitive constraints are built over variables, domains, arithmetical operators in {+,-,*,\,…} and relations {>,≥,=,≠,≤,<} while non-primitive constraints include user-defined constraints and constraint operators that express high-level relation between other constraints. In this paper, we define constraint operators that model dynamic memory allocation, deallocation, accesses and updates. These constraints apply deduction rules as any other constraint of a finite domain constraint solver.

Two interleaved processes intervene in the solving process of a finite domain constraint system: constraint propagation and variable labeling.

2.1.1 Constraint propagation

Initially, the constraints are added to a main queue and fall into in an ***evaluation*** state. Each constraint of the queue is considered one-by-one by the constraint propagation algorithm. The algorithm exploits each constraint to filter out the inconsistent values from the domain of the variables. When the domains of all variables of the constraint have been pruned, the constraint falls into the ***suspended*** state. When the domain of a variable is pruned, other constraints that involve this variable are reintroduced into the queue. In this case, these *suspended* constraints are woken up and return in the *evaluation* state. If the domain of at least one variable becomes empty, the constraint ***fails***: the constraint system is unsatisfiable. If the constraint succeeds meaning that each of the tuples from the current domains are compatible with the constraint, then it falls in the ***entailed*** state. In this case, the constraint is removed and is not considered anymore in the process since it is no more useful. When no more reduction is possible, the queue becomes empty and the constraint propagation ends.

2.1.2 Variable labeling

When the constraint propagation ends, enumeration on the possible values from the domains is usually required to get a solution. The labeling procedure tries to give a value to every variable one by one. When a value is chosen from the domain of a variable, the constraint propagation is re-run to prune the domains of other variables with the current hypothesis. If a contradiction appears during the resolution process, the procedure backtracks to other possible values. The process stops when a value is assigned to each variable.

2.2 Notations, syntax and restrictions

2.2.1 Notations

In the rest of paper, we will use capitals, possibly subscripted, to denote finite domains variables and variable of the constraint model (such as memory for example). On the contrary, we will use lower-case letters to denote program variable and will use a special operator, noted @, to denote addresses of the memory.

2.2.2 Syntax and restrictions

```
program      ::= { statement* }
statement    ::=
   assignment
  | malloc(type_size)        %memory allocation
  | free(expr)               %memory deallocation
  | if( expr ) { statement* } else { statement* }
  | while( expr ) { statement* }
assignment   ::=
   var = expr | *var = expr | var->f = expr

expr ::=
   cte | var | &var | *var | var->f | expr + expr | …
```

Figure 2. Syntax of the pointer language

For the sake of clarity, we confine ourselves to a small C-like language (whose an excerpt of the grammar is given in Fig. 2) that includes pointers assignment and dereferencing, structures management, memory allocation and deallocation. This language is powerful enough to express any computable function and possesses all the necessary features to deal with dynamic allocated structures. However, it also presents a lot of restrictions w.r.t. the C language. In particular, it does not allow unstructured code, type casting, unconstrained pointer arithmetic, volatile variables, function calls, etc. In this paper, we focus on the problem of dynamic memory management within our constraint-based test data generation method, hence we will only detail the constraint model for the operators on memory and will left apart the rest. Similarly, we will only give an overview of the complete test data generation method. The interested reader can consult ref [4] to see precisely how we deal with control structures (conditionals, loops), ref [5] to understand the problems of pointer aliasing in constraint-based approaches, and ref [6] to understand how we deal with floating-point computations.

3. Goal-oriented test data generation based on constraint solving

3.1 Constraint generation

As said previously, CBT involves two processes: constraint generation and constraint solving. In our framework, the constraint generation step involves the translation of a program under test into a constraint model over finite domain variables and memory variables. This constraint model can be seen as a relational formulae over the input memory state of the program and its output memory state. Note that such memory states can contain unknown references such as input pointer formal parameters or unknown integer variables. In fact, there is no strong hypothesis on the context of call and each C function is treated in isolation in our model. Moreover, our goal-oriented approach prevents using path expression to remove ambiguities. As soon as a conditional or a loop is encountered, our model can contain unknowns due to the non-determinism of the control flow. Consequently, we need a constraint model able to deal with unknown parts of the memory. We get this model by a systematic and inductive translation of statements into constraints. We will describe this model on a simple but illustrative example. Consider the following sequence of statements:

```
      C code                                  Constraint model

 a.   i = i+1          ------------>     load_elementt(M1, @i, I1)
                                         I2 = I1 + 1
                                         store_element(M1, M2, @i, I2)

 b.   if( i > 10 )     ------------→      load_element(M2, @i, I3)
                                          B1 = (I3 > 10)
                                           ite(B1,
 c.     { p = &i;}                              store_element(M2, M3, @p, @i)and M4 =
M3,
                                              M4 = M2
                                               )

 d.   *p = *p + 5      ------------>     load_element(M4, @p, P)
                                         load_element(M4, P, DP1)
                                        DP2 = DP1 + 5
                                         store_element(M4, M5, P, DP2)

 e.   if( i < 16 ) …   ------------→    load_element(M5, @i, I4)
                                         B2 = (I4 < 16)
                                         ite(B2, …
```

The C code presents an aliasing problem at statement d as we ignore whether p points to i or not at this point (suppose for example that i and p are input formal parameters of the program under test). Statement `a` is translated into three independent constraints. The first one states a relation between the (abstract) memory `M1` and the variable value `I1` that is associated to program variable `i` through its reference `@i`. This relation holds anytime an access to program variable `i` is made, but depends on the current state of the memory at a given point of the program. The second constraint states a relation between two finite domain variables (`I1` and `I2`) where `I2` is a new fresh variable. Our constraint model does not require variables to be declared. Finally, the third constraint links memories `M1` and `M2` (two states of the same abstract memory), and variable `I2`: the new state of `M2` should be the update of the state `M1` with `I2` at reference `@i`. This decomposition shares similarities with a classic 3-

address code that can be found in many compilers. Statement `b` is a conditional and we use an auxiliary boolean variable `B1` to associate with the truth value of the decision `i > 10`. The conditional itself is translated to a special constraint operator in our constraint model noted `ite(B1, Then, Else)`. Loops are equally treated with a special constraint operator called `w`, see [4,7] for more details. Note that, when no deduction is possible on the path that should be followed, the conditional operator makes the union of the two possible memory states. In our example, M2 and M3 joint in the memory state M4. M3 is the memory state obtained after interpretation of the constraints of the Then-part of the conditional while M2 is just the memory available if the Else-part is taken (no statement in this example). This way of interpreting conditional has much to do with the Static Single Assignment form of imperative program [4]. Statement `c` is translated into a single `store_element` constraint that links memory `M2` and `M3`. Statement `d` is translated into four constraints. The two first permit to access to the value stored in memory `M4` at reference `*p`, while the two latter permit to store the result of expression to memory `M5`. The need for `load_element` to be a constraint relation (as opposed to a function) is clearly stated at statement `e`. As the value of `p` is determined by the control flow, we have to get it at the memory state `M5` which is unknown or only partially known before having selected a path through the conditional. One can see here how pointer aliasing is handled through the use of special constraint operators. This also appeals for powerful deduction rules in case some information is available on memory states. As a result, this constraint model allows us to implement goal-oriented test data generation in a constraint-based approach.

3.2 Constraint solving

The constraint solving step aims at finding an input memory state that satisfies a given testing objective. Consider the objective of generating a test data that reaches the Then-part of statement `e`. By using the control dependencies within the program, this testing objective is directly translated into the constraint `B2 = 1`, forcing I4 to be strictly less than 16 and so constraining the memory variable `M5`. By applying the deduction rules of each constraint operator of the model in a constraint propagation algorithm, our system makes the non trivial deduction that the Then-part of statement `b` cannot be executed. In fact, if the program variable `i` at statement `b` is strictly greater than 10 then `*p` and `i` would be aliased at statement `d`, and the value of `i` would be greater than 16 at statement `e`, which is contradictory with the testing objective. As a consequence, our system deduces that the value of `i` at the beginning of the program has to be less than 9. Thanks to the relational view of each statement, constraint reasoning is possible and deduces interesting facts about the input state of the program in order to satisfy the selected testing objective. Test data generation is obtained just by launching a labelling search that chooses i to be zeroed in the input state of the memory.

It is worth noticing that our constraint reasoning did not exclude the possibility that p points to i in the input state of the memory. Such input aliasing relationship depends on the context of call.

4. Description of an abstract memory

In our system, we express C operations over the memory with relations over two (abstract) memory states. The first memory state represents the memory before statement execution while the second memory state represents the memory after statement execution. An **abstract memory** in our constraint system is an abstract model of the physical memory at a given program point and contains information known *at a given step of the resolution process of the overall constraint system*. Figure 3 describes the content of an abstract memory. m : it contains four elements: struct(m), tab_i(m), tab_p(m), and closed(m) that are described in the following subsections.

Description of sets :
MEM: set of abstract memories
$IDENT$: set of locations in abstract memories
VAR_i: set of abstract integer variables
VAR_p: set of abstract pointer variables
VAR_s : set of abstract structures
$TYPE_s$: set of structured types
$NAME_f$:set of names of fields
$TTPE$: set of types. $Type = \{integer, pointer\} \cup Type_s$

$\forall m \in MEM : m =< tab_i(m), tab_p(m), struct(m), closed(m) >$
with: $tab_i(m) \in \mathcal{P}(\{< ident, V_i > | ident \in IDENT, V_i \in VAR_i\})$
$tab_p(m) \in \mathcal{P}(\{< ident, V_p > | ident \in IDENT, V_p \in VAR_p\})$
$struct(m) \in \mathcal{P}(\{< type, V_s > | type \in TYPE_s, V_s \in VAR_s\})$
$closed(m) \in true, false$

$s\_@ : \ VAR_s \rightarrow \mathcal{P}(IDENT)$ % set of program locations
$access\_f : IDENT \times NAME_f \rightarrow IDENT$ % access to a field
$dom_i : VAR_i \rightarrow \mathcal{P}(\mathbb{N})$
$dom_p : VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL) \cup \{all\}$
$ndom_p VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL)$
$type_p : VAR_p \rightarrow TYPE$

Figure 3. Abstract memory

In order to facilitate the understanding, we provide an example of abstract memory. Figure 4 shows the abstract memory state m obtained before statement 11 of the Josephus program after two iterations of the first loop.

$tab_i(m)$

Ident	Var_i	dom_i
m	V_i1	Inf..sup
n	V_i2	2
i	V_i3	3
n(2).key	V_i4	1
n(7.1).key	V_i5	2
n(7.2).key	V_i6	3

$tab_p(m)$

Ident	Var_p	dom_p	non_dom_p	$type_p$
t	V_p1	{n(7.2)}	empty	node
x	V_p2	{n(2)}	empty	node
n(2).next	V_p3	{n(7.1)}	empty	node
n(7.1).next	V_p4	{n(7.2)}	empty	node
n(7.2).next	V_p5	all	empty	node

struct(m)=<node, S_{node} >

Var	s_@
S_{node}	{ n(2), n(7.1).n(7.2)}

closed(m)=true

Figure 4. Abstract memory after the statement 11 of the Josephus program, at the end of the constraint propagation when the first loop is enrolled twice

4.1. Structures dynamic allocations: struct(m)

For an abstract memory m at a given step of the solving process, information about known (statical or dynamical) allocation of structures is stored in struct(m). In a memory, a variable is associated to each structure type definition. In figure 4, the variable S_{node} is associated to the type node. A function called s_@ gives the set of anonymous program locations associated to this variable. On the model, we represent each abstract memory location with the term ident. An abstract memory location can be anonymous in the case of dynamic allocation. In figure 4 where there are three dynamically allocated structures, *{n(2), n(7.1), n(7.2)}* is the set of the anonymous program locations. For example, *n(7.2)* denotes the anonymous program location obtained by the execution of the statement 7 in the second iteration of the loop. The set given by s_@ can only be enlarged during the resolution process ot the constraint system. Function access_f associates to location loc of a structure and field name f, the complete name of the field location loc.f. For example, *access_f(n(1),next)* is *n(1).next.*

4.2. Basic types: $tab_i(m)$, $tab_p(m)$

Information about basic variables is memorized by pair ident-Var; where ident is an abstract memory location, and Var is a variable of type integer or pointer. On the model, the sets of integer and pointer variables are noted respectively VAR_i and VAR_p. These pairs are stored in two data structures, called *tableaux*: $tab_i(m)$, $tab_p(m)$. The number of pairs contained in the *tableaux* represent the known integer or pointer locations and therefore can only increase or remain the same during the resolution process of the constraint system..

4.2.1. Integer variables. For abstract variables that represent integers, the function dom_i gives the set of possible values at a given step of the resolution. For example, the abstract memory location i in $tab_i(m)$, has value 3 in abstract memory of figure 4.

4.2.2. Pointer variables. For abstract variables that represent pointers, our model provides two functions dom_p and $ndom_p$. dom_p returns the set of possible abstract memory locations (symbolic names or anonymous program locations) for the pointer while $ndom_p$ returns the set of memory locations that cannot be pointed by the pointer. For example, on a condition such as (p==&a||p==&b), we get $dom_p(P)$={@a,@b} where @a (resp. @b) denotes the address of a (resp. b) in the abstract memory model. $ndom_p$ is usefull in the case when dom_p=all. dom_p=all means that p can point to any location in the memory except the locations contained in $ndom_p$. On a condition such as (p!=&a) there are two possible changes in dom_p and $ndom_p$. If dom_p=all, then @a is added to $ndom_p$. Otherwise, @a is removed from dom_p. These deductions on pointer domains are expressed by the notation $P \neq$ @a in the description of the operators in section 5. If $dom_p(P)$ contains a single value v, the element pointed by P is definitely known. It is noted $P=v$.

For example, *Vp2* in figure 4 is simplified to *n(2)* as x points to the first dynamically allocated structure in statement 2.

The function $type_p$ returns the type of the pointed element. In figure 4, all the pointers point to a node structure. The number of elements in Dom_p can only decrease or remain the same during the resolution process of constraint system.

4.3. Closure of the memory: closed(m)

The last component of an abstract memory m is the predicate closed(m). It indicates whether all the elements of the *tableaux* and structures are defined or not in the current abstract memory. If closed(m) is false for an abstract memory at a step of the resolution process of the constraint system, it can become true later. If closed(m) is true at a step of the resolution process, then it remains true during all the process. The labeling process uses this predicate to force the input domain to contain only the data structures previously labeled. Without this predicate, labeling could invent new structures and values leading to a non-terminating process. Such status is of particular interest when one looks for the possible shapes of a dynamic structure during the labeling process as it strongly constrains the set of identifiers that can be pointed to by a pointer.

5. Constraint operators on abstract memories

To find a test datum to reach a given testing objective, the program under test is translated into a constraint system on abstract memories. This section details the deduction rules applied within the constraint operators that tackle with dynamic structures. Figure 5 shows the translation of some basic statements of our language into these constraint operators. M_{in} and M_{out} are the memories respectively before and after a given statement. All the elements of the memory M_{in} and M_{out} but the input parameters of the constraint operators are identical. Asterisks in figure 5 denote the operators that are detailed in the following. For other operators, we will only give an overview.

new(t,id)
(*) new_s(Struct(M_{in})(t), Struct(M_{out})(t),id)
new_fields(M_{in},M_{out},t,id)

x=y->f
(*) load_element(Tab_p(M_{in}),y,Y)
(*) access_s(Struct(M_{in})(typeY),Y,f,V_p)
(*) load_element(Tab_t (M_{in}),Vp,Val)
(*) store_element(Tab_t(M_{in}), Tab_t(M_{out}),x,Val)

y->f=x
(*) load_element(Tab_p(M_{in})),y,Y)
(*) access_s(Struct(M_{in})(typeY),Y,f,V_p)
(*) load_element(Tab_t (M_{in}),x,X)
(*) store_element(Tab_t(M_{in}), Tab_t(M_{out}),Vp,X)

x=y
(*) load_element(Tab_t(M_{in}),y,Y)
(*) store_element(Tab_t(M_{in}), Tab_t(M_{out}),x,Y)

free(x,t)
(*) load_element(Tab_p(M_{in}),x,X),
(*) delete_s($Struc_t$(M_{in})(t), Struct(M_{out})(t),X)
delete_fields(M_{in},M_{out},type,X,)

Figure 5. translation into operators

Statement **new(t,id)** generates two operators. The first one links the set of locations of type t between M_{in} and M_{out},. It needs the identifier id of the location to add. The second constraint operator reserves place for the fields of the new structure. The statement **x=y->f** generates four operators. The first one loads the value of y. The second one gets a pointer Vp that points to the possible locations of y->f. Thanks to Vp, the third operator collects in Val the possible values for y->f. t in the notation Tab_t (M_{in}) means that the tableau from which the value is loaded depends on the type of y->f (integer or pointer). The fourth operator sets the domain of Val as domain of possible values for x. Similarly, statement **y->f=x** generates four operators. The first one loads the value of y. The second one gets a pointer Vp that points to the possible locations in memory that can store y->f. The third one loads the value of x. The fourth one affects the value of x to y->f. Statement **x=y** is expressed by two operators. The first one loads y. The second operator stores y in the location of x in the abstract memory. Statement **free(x,t)** generates three operators: the first one loads the value of the pointer x. The second one deletes the structure pointed by x in the memory while the third one deletes its fields.

We now turn on the description of these constraint operators: new_s for allocation, delete_s for deallocation, access_s for structure field access, store_element to store integer/pointer values in memory, and load_element to load values. The constraint operators for dynamic memory management have three roles: the propagation of the knowledge of the allocated locations in the memories, the filtering of the domains for pointers and integers values and the propagation of information about the closure of the memory.

5.1. Representation of a constraint operator

We propose to explain our constraint operators by using a simple model based on finite state machines. Figure 6 shows a generic state machine for constraint.

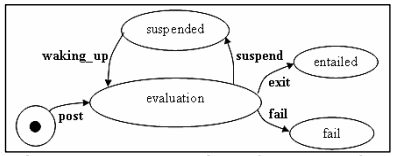


Figure 6. representation of a constraint operator

Our representation gives the possible states of a constraint: once posted, it can be in evaluation, suspended, entailed or in a failure state. Labels on the arrows describe the events that permit to switch from one state to another. For each operator, we describe five possible transitions: 1) post event: the operator is posted in the propagation queue ; 2) waking-up event: the operator is woken up by some additional information on the domain of its variables or on its status ; 3) suspend event: no more deduction rules can be exploited to prune the variation domains ; 4) exit event: the operator becomes entailed ; 5) fail event: some inconsistency has been detected which indicates failure of the current constraint system.

Figure 7 shows a C program that illustrate the interest of the deduction rules that we are going to present. In the following, we

```
    a= new(t,new(1)) ;
M0
    b= new(t,new(2)) ;
M1
    c= new(t,new(3)) ;
    a-> f =1;
    b-> f =2;
    c->f =3;
    if(cond1){
       p=a;
    }else{
       if(cond2){
          p=b;
       }else{p=c}}
    if(cond3){
M3
       free(p);
M4
    }else{
M5
       p->t=i;
M6
       j=p->t;}
    if(p!=a && b->t=6 && j>2){
```

Figure 7. program foo

will refer to this program implicitly by showing only the abstract state of the memories M0,..,M6.

5.2. The new_s operator

The operator new_s(S0,S1,id) is added whenever a structure of type t is dynamically allocated. Figure 8 shows the model of this operator that establishes a relation between S0, S1 and id. In the model, we suppose that

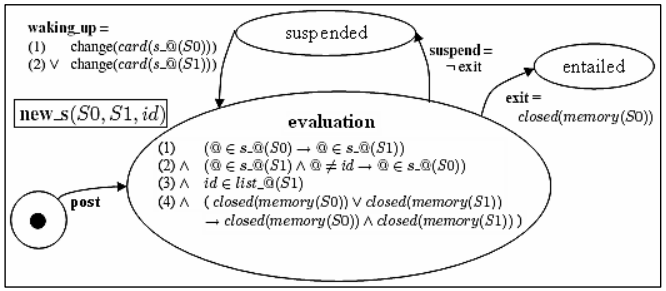


Figure 8. new_s operator

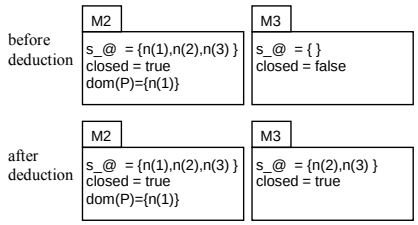
S0=struct(M_{in})(t), S1=struct(M_{out})(t), where struct(M)(t) denotes the variable associated with t in struct(M), and id is the identifier of the anonymous dynamic location. The operator is awoken when a location is added to the set of locations of S0 or S1.

If the operator is awoken, some new deductions can be performed. S0 and S1 should contain the same identifiers, except for id that is only in S1. Firstly, S1 should contain all the locations that are present in S0 (proposition 1) as well as the location id (proposition 3). Secondly, all the locations that are in S1, except id, should be in S0 (proposition 2). Moreover, if one abstract memory is closed, the second one should also be closed (proposition 4). Indeed, as new_s adds only one new location, the closure of M_{in} (resp. M_{out}) implies the closure of M_{out} (resp. M_{in}). Moreover, the operator succeeds as soon as M_{in} is closed (cf exit arrow), while it suspends otherwise.

5.3. The delete_s operator

The operator delete_s(S0,S1,X) is added whenever a structure of type t, pointed by X, is removed from the abstract memory. Figure 9 illustrates the delete_s operator, which maintains a relation between three parameters: S0=struct(M_{in})(t), S1=struct(M_{out})(t), and X.

The operator delete_s is woken up either when a location is added to the set of locations S0 or S1, or when the variation domain of pointer X is modified (for example, learning that X points to only one location). Such a modification is noted change(X) in our model. The relation maintains the fact that X should be non-null (proposition 1). As an illustration of the deduction rules, suppose the information on abstract memories linked by a delete_s operator shown below is available (memories before deduction).



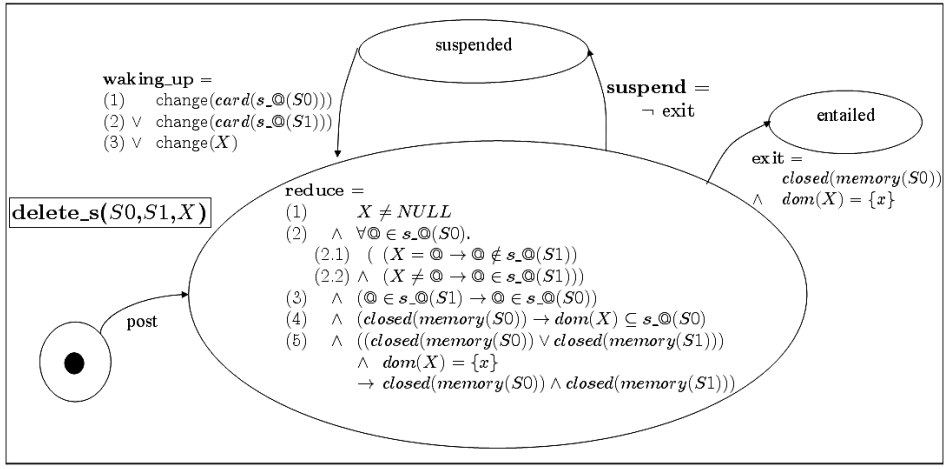


Figure 9. delete_s operator

Here, P points definitely to n(1) which is the deleted location. So, it should not appear in M3 (proposition 2.1) and other locations are not touched (proposition 2.2) and must appear in M3. Moreover, as M2 is closed and the location to delete is known, M3 is also closed (proposition 5) and the operator succeeds (exit arrow). We obtain the memories after deduction shown above. If the input memory is closed, then X can only point to a location contained in the set of addresses of S0 (proposition 4).

If there is no precise information on the pointed location or the available memory, then the operator is suspended.

5.4. The access_s operator

The operator access_s(S,X,f,Vp) is added when we access to a field of a structure of type t. Figure 10 illustrates this operator that maintains a relation between four parameters: S=struct(M)(t), X the pointer to the possible locations of the structure, f the field name and Vp the pointer to the possible locations of the field.

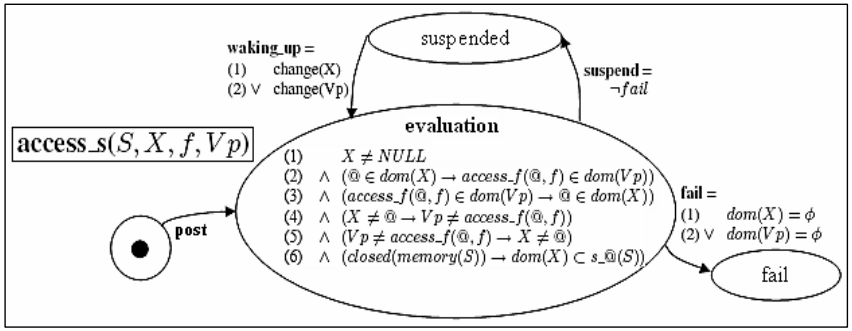


Figure 10. access_s operator

The operator access_s is awoken when we get more information about the values pointed by X or by Vp. X should not be null (proposition 1). Statement `p->t=i;` in figure 7 leads to add two constraint operators including access_s(struct(M4)(t),P,t,Vp). The following draw illustrates the deductions made by this operator. For each location of the domain of P, the relation maintains that Vp can point to the location associated with the field (proposition 2).

before deduction

M4
dom(P) ={n(2),n(3)} dom(Vp)=...

after deduction

M4
dom(P) = {n(2),n(3)} dom(Vp)={n(2)->t,n(3)->t}

In the following example, Vp can only point to n(2)->t or n(3)->t so P can only point to n(2) or n(3), n(1) is removed from its domain. (proposition 5).

before deduction

M4
dom(P) = {n(1),n(2),n(3)} dom(Vp)= {n(2)->t,n(3)->t}

after deduction

M4
dom(P) = {n(2),n(3)} dom(Vp)={n(2)->t,n(3)->t}

Proposition 6 gives complementary information: if the memory M is closed, all the possible locations pointed by X belong to the set of addresses of S. Indeed all the possible locations for a structure of type t are known and X points to such a structure.

During the resolution process, if the domain of a variable becomes empty, the constraint resolution process fails. Indeed, it means that there is no assignment for all the variables of the system such that all the constraints are satisfied. For the operator access_s, the only domains that can become empty are the domain of X and the domain of Vp (in this case X or Vp cannot point to any location anymore).

5.5. The store_element operator

The operator store_element(Tab(M_{in}),Tab(M_{out}),X,V) is added when the statement stores a value in memory at a given address. Figure 11 illustrates this operator that maintains a relation between X, Tab(M_{in}), Tab(M_{out}) and V. X is a pointer to a location that stores an integer or a pointer value, V is the value to store.

The store_element operator is awoken when 1) a pair <ident,Var> is added in one of the *tableaux* Tab(M_{in}) and Tab(M_{out}) (conditions 1 and 2) ; 2) when the domain of a variable in Tab(M_{in}) or Tab(M_{out}) changes (condition 3); 3) when the information about the pointer X changes; 4) when the domain of the value V to store changes.

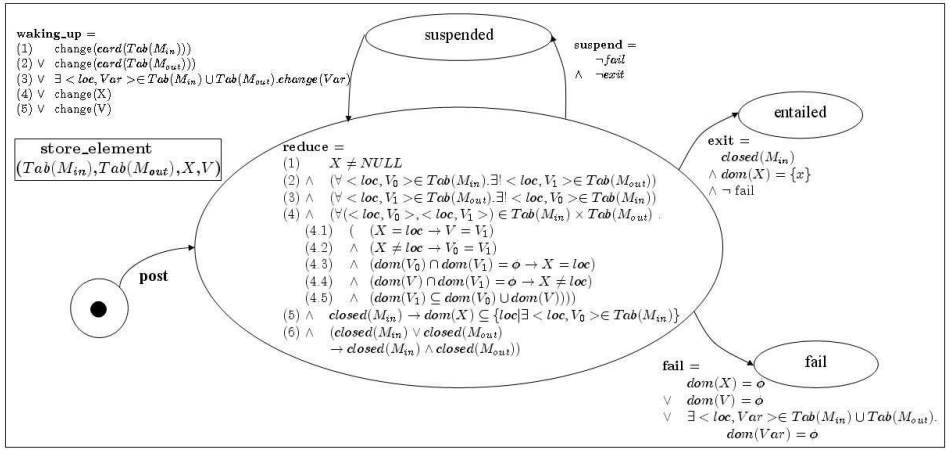
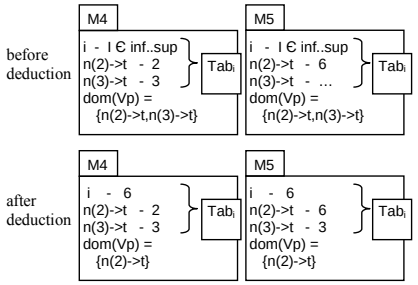


Figure 11. store_element operator

X should be non-null (proposition 1). Consider again statement `p->t=i;` in figure 7 and let Vp be a variable that points to the possible locations of p->t. The storage of the value of I in the location pointed by Vp is performed within the relation maintained by the store_element operator.

In the figure below, in the memories before deduction, as domains of n(2)->t before and after the storing statement are distinct the value of n(2)-> t is changed by the storing statement. It means that the value of I is stored in n(2)->t. As a consequence, Vp points to n(2)->t (proposition 4.3). dom(n(2)-> t) in M5 and dom(I) should be intersected in order to find the values of I and n(2)->t in M5 (proposition 4.1). The domain of n(3) -> t remains the same in both memories (proposition 4.2). We obtain the following memories:



Other deductions associated with the operator include the following rules:

- Any couple <loc,Var> existing in one of the two memories should also appear in the other memory (propositions 2 and 3);
- For all the pairs (<loc,V_0>,<loc,V_1>) in Tab(M_{in}) x Tab(M_{out}):

- If dom(V)∩dom(V_1)≠∅, the variable V cannot be stored at the location loc, so X ≠ loc (proposition 4.4)
- In other cases, we can deduce that dom(V_1) is included in dom(V_0) U dom(V) (proposition 4.5)

The solving process fails if the domain of X, or the domain of an abstract variable stored in M_{in} or M_{out}, or the domain of V becomes empty.

After constraint propagation, store_element succeeds if M_{in} is closed and the value pointed by X is known. Indeed, in this case all the information that permits to deduce the contents of Tab(M_{in}) and Tab(M_{out}) is available.

5.6. The load_element operator

The operator load_element(Tab(M),X,V) is added when the program loads an integer or pointer value from the memory at a given address. Loading a value does not modify the memory so it constraints only a single memory. Figure 12 illustrates the operator that maintains a relation between X, V and M, where pointer X points to a variable V in the corresponding *tableau* of M.

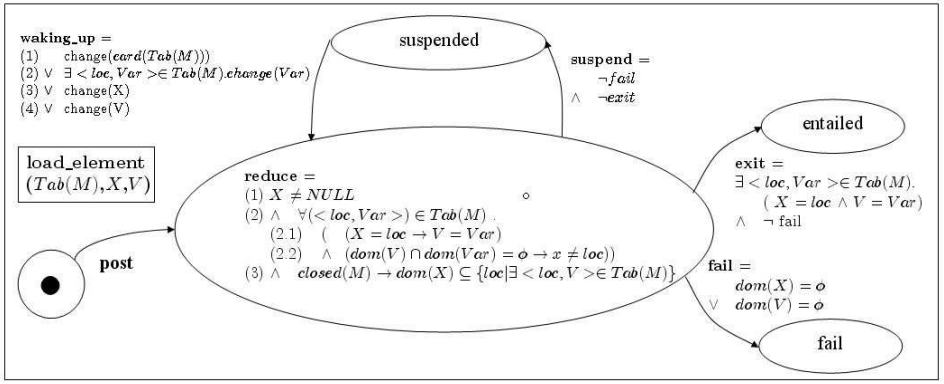


Figure 12. load_element operator

Proposition 2 expresses that for all the pairs <loc,V_{ar}> in Tab(M):

- If X points to loc, the variable V is loaded from the location loc and V=V_{ar}. The domain of V and V_{ar} is then dom(V)∩dom(V_{ar}).

- If $dom(V) \cap dom(V_{ar}) \neq \emptyset$, the variable V cannot be loaded from the loc location and then $X \neq loc$.

The solving process fails if the domain of X or V becomes empty, while it succeeds if there is a pair $<loc, V_{ar}>$ in Tab(M) such that X=loc and $V=V_{ar}$.

6. Experimental results on the Josephus program

We implemented the operators that are described above. Our system is able to take a program written in a restricted syntax of the C language and generates automatically test data w.r.t. some testing objectives. The system is developed in C and Prolog and follows the principles of the previous implementation INKA.

As an illustration of the efficiency of our operators, we generated test data for the Josephus program in figure 1. We considered several testing objectives. Among them, we generated a test suite that covers all the branches of the program in less than 1 msec of CPU time. The results were computed on an Intel Pentium, 2.16 GHz machine running Windows XP with 2.0 GB of RAM. To cover this objective, INKA first tries to find a test case to reach the deepest instructions of the control flow graph. In the case of the Josephus, it means that it tries to enroll the loop **while3** at least once. The test suite generated contains {(3,2)} as values for m and n. The output memory shape obtained with our model is in accordance with the one obtained by executing the program, which shows that the operators faithfully model dynamic memory management.

We also dealt with more complex requests as the one or reaching four iterations of **while3** in the first iteration of **while2**. It is worth noticing that reaching this testing objective is hard, as witnessed by the fact that a random test data generator will probably never achieve to reach it. In fact, the likelyhood of drawing a test data (values for n and m) such that this objective would be covered is not far from zero, as there is only a single value for m able to satisfy it. Thanks to our constraint reasonning on operators, we obtained the test case m=5, n=2 in 109 msec. If we take into account the wide domain for m and n as input values, the probability to reach this objective with random test case generation is low.

Regarding the objective of unrolling k times the loop **while2**, which is the objective described in introduction of this paper, we obtained the results shown in the curve on figure 13. The test cases obtained are of the form m=0, n=k+1. When k is less than 15, the generation of a test case to reach the objective takes less than 5 sec. We claim that these results are promising because they confirm the high deductive potential of our approach. Nevertheless, as shown in the curve, runtime increases exponentially with the value of k. Indeed, the number of operators to handle dynamic memory management in the constraint system increases with k. So the number of constraint waking up, costly in time, increases with the number of operators.

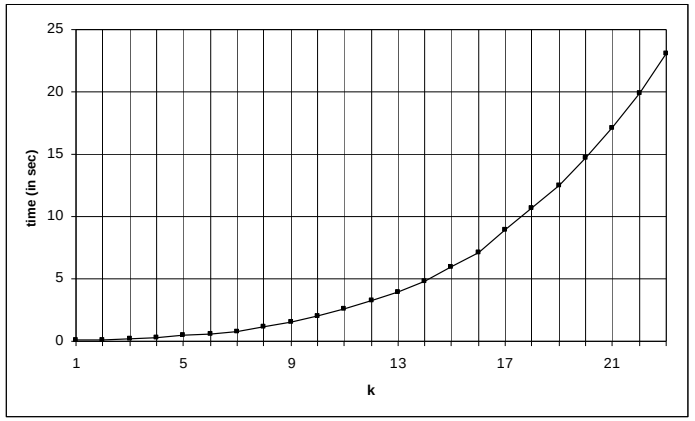


Figure 13. runtime for the generation of a test case reaching k iterations of while2 in the Josephus program

7. Related Works

Dynamic test data generation exploits program executions to find test data that satisfy a testing objective [20]. In this process, program executions guide the search towards the next test data by optimizing a cost function called the branch function. As the search is only based on concrete executions, this process handles dynamic data structures. Visvanathan and Gupta studied in [21] the problem of generating test data for functions with pointer inputs. They proposed to enumerate the possible data structures shapes by exploiting an address table symbolically computed during the execution of a program path. Sai-ngern et al. [22] followed a similar path by managing the table with a dynamic linear array. Unlike these approaches, our method is symbolic and it does not depend on execution trials. Symbolic test data generation is usually considered as more powerful to deduct information and our operators that model dynamic memory management are equipped with deduction rules in order to prune early the search space of possible about data structures shapes.

Williams et al. [9] and Sen et al. [11] address the problem of generating test data for C functions with dynamic structures by using symbolic execution and constraint solving techniques. In their approaches, constraints on input values permit to handle pointer relationships and aliasing problems occur only within input data structures. In our approach, we have proposed dynamic memory management operators able also to deal with pointer aliasing problems that are located in the source code. PathCrawler [9] and CUTE [11] are two test data generators trying to cover all the feasible paths of C programs. Both systems try to solve path conditions in order to find the next test data that will follow a path that improve the current coverage of the program. These tools are path-oriented, meaning that they require a path to be selected first. Unlike path-oriented and among other advantages, goal-oriented methods

such as the one presented in this paper, exploit the early detection of non-feasible paths to prune the search space made up of all the paths that reach a given branch. Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths can be exponential on the number of decisions of the program or even infinite when loops are unbounded. Moreover, thanks to the constraint reasoning on operators, our implementation permits to express more complex testing objectives such as reaching a selected point at certain iterations of a while loop. This is particularly interesting for programs that build dynamic data structures as such requests help verifying their shapes during testing. However, one disadvantage of our approach is that it requires the constraint solver to be adapted and modified which prevents the usage of some commercial solvers that are sometimes more efficient.

8. Conclusion and perspectives

In this paper, we presented a new constraint-based model that handles dynamically allocated data structures in goal-oriented test data generation. Our model is built over specific constraint operators that are equipped with powerful deduction rules. These operators handle dynamic memory allocation, deallocation, loading and update of pointed structures. We implemented these operators within INKA a test data generator for programs written in a restricted subset of C. This implementation was challenging because it required building a new constraint solver over pointer and memory variables. However, our implementation still suffers from some restrictions. It is based on a memory model that does not include physical information about variables. For example, size of data types and bit vectors are not considered in the model and then, data structures such as unions or bit fields and physical type casting cannot be handled. Function pointers have also been left apart for the moment. Our goal-oriented approach is based on testing objectives that specify statements or branches in structured programs only and then goto statements are not currently handled. For all these reasons, we consider that the subset of the C language currently handled by our tool is too tight to be practically useful on real-world programs. But before evaluating our model on larger programs, we wanted to be sure that the approach was suitable and efficient on small but complex programs. Thanks to our constraint model, we successfully generated test data that cover complex testing objectives for the C program Josephus, which involves the creation and destruction of circular linked lists.

Our future work will be dedicated to extend this approach to inter-procedural test data generation and to a larger subset of the C language. Dealing with function calls and dynamic memory allocation requires paying attention on how constraint systems are built as the number of constraints can grow exponentially with the number of function calls. Hence, just inlining function calls will not be an acceptable solution and we would probably need some kind of abstractions. Dealing with unstructured code will also be a real challeng as our approach builds over constraint operators that model control structures (conditionals, loops) and goto statement usually break the flow. In fact, modelling exactly the semantics of such constructions is difficult and our line of work will be focussed on the possible combination of constraints and abstractions to approximate the behaviours of these statements in constraint-based automatic test data generation.

9. References

[1] DeMillo, R. A. and Offutt, A. J. 1991. "*Constraint-Based Automatic Test Data Generation*". IEEE Trans. Softw. Eng. 17, 9 (Sep. 1991), 900-910.

[2] Jeremy Dick and Alain Faivre "*Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*" *Proc. Of Formal Methods Europe* (FME 1993), pages 268-284

[3] Bruno Marre, "*Toward Automatic Test Data Set Selection Using Algebraic Specifications and Logic Programming*" *in Proc. of the Int. Conf. on Logic Programming* (ICLP 1991), pages 202-219

[4] Gotlieb, A. , Botella, B. and Rueher, M., *"Automatic Test Data Generation Using Constraint Solving Techniques"*, *in Proc. Of the Int. Symposium on Software Testing and Analysis* (ISSTA 1998), Clearwater Beach, FL, USA, 1998

[5] Gotlieb, A. and Denmat, T. and Botella, B., *"Goal-oriented test data generation for programs with pointer variables"*, *in Proc. of the 29th IEEE Annual International Computer Software and Applications Conference* (COMPSAC 2005), Edinburh, Scotland, 2005, pp. 449-454

[6] Botella, B. and Gotlieb, A. and Michel, C., *"Symbolic execution of floating-point computations"*, The Software Testing, Verification and Reliability journal 16 (2), John Wiley, 2006, pp 97-121

[7] Gotlieb, A. and Botella, B. and Watel, M., *"Inka: Ten years after the first ideas"*, *in Proc. of the 19th International Conference on Software and Systems Engineering and their Applications* (ICSSEA 2006), Paris, France, 2006

[8] Meudec, C., "*ATGen: automatic test data generation using constraint logic programming and symbolic execution"*, The Software Testing, Verification and Reliability journal 11 (2), John Wiley, 2001

[9] Williams, N., Marre, B., Mouy P. and Roger, M. *"PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis" In Proc. of the 5th European Dependable Computing Conference* (EDCC 2005), Budapest, Hungary, LNCS Vol. 3463/2005, Springer-Verlag, 2005, pp 281-292

[10] Godefroid, P., Klarlund, N., and Sen, K. 2005. *"DART: directed automated random testing"*. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2005), Chicago, IL. pp 213-223

[11] Koushik Sen and Darko Marinov and Gul Agha, *"CUTE: a concolic unit testing engine for C"*, *In Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (ESEC/FSE-13), ACM Press, Lisbon, Portugal, 2005,pp.263-272

[12] D. Lewin and L. Fournier and M. Levinger and E. Roytman and G. Shurek, *"Constraint Satisfaction for Test Program Generation"*, *in Proc. Of the IEEE International Phoenix Conference on Communication and Computers*, (IPCCC 1995) Phoenix, 1995

[13] Bin, E. and Emek, R. and Shurek, G. and Ziv, A., *"Using a constraint satisfaction formulation and solution techniques for random test program generation"*, IBM Systems Journal 41 (3), 2002

[14] Daniel Jackson and Mandana Vaziri, *"Finding bugs with a constraint solver"*, *In Proc. of the ACM SIGSOFT Iinternational Symposium on Software Testing and Analysis* (ISSTA 2000), pp. 14-25, Portland, Oregon, United States, 2000

[15] C. Pasareanu, M. Dwyer and W. Visser, *"Finding Feasible Abstract Counter-Examples",* STTT Journal, 5, (1),2003

[16] Collavizza, H. and Rueher, M., "*Exploration of the Capabilities of Constraint Programming for Software Verification"*, *In Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'06), Vienna, Austria, 2006

[17] Ferguson, R. and Korel, B. 1996. *The chaining approach for software test data generation*. ACM Transactions On Software Eng. And Methodology. 5, 1 (Jan. 1996), 63-86

[18] Sedgewick, R. "*Algorithms in C"*, Addison-Westley publishing company, 1988

[19] Utting, M. and Legeard, B. "*Practical Model-Based Testing: A Tools Approach*" Morgan Kaufmann, 2007

[20] Korel, B. "*Automated Software Test Data Generation*," IEEE Transactions on Software Engineering ,vol. 16, no. 8, pp. 870-879, August, 1990.

[21] Visvanathan, S., Gupta, N. "*Generating Test Data for Functions with Pointer Inputs,*" *in Proc. Of the 17th IEEE Automated Software Engineering Conference (ASE'2002)*, pp. 149-160, 2002.

[22] Sai-ngern, S. and Lursinap, C. and Sophatsathit, P., *"An address mapping approach for test data generation of dynamic linked structures",* Information and Software Technology (47), 2005,pp. 199-214

Chapter 6

Modélisation à contraintes des constructions orientées-objet

L'article de ce chapitre présente une méthode et un outil de génération automatique de données de test pour les programmes en Bytecode Java, qui s'appuie sur le modèle–mémoire donné dans [Charreteur 09] au chapitre précédent. Bien que bénéficiant de l'expertise de nos collègues de l'équipe-projet INRIA Lande puis Celtique, plusieurs difficultés techniques sont apparues lorsque nous avons voulu étendre notre approche à base de contraintes au cas des constructions orientées-objet pour le Byecode Java. Tout d'abord le traitement du flot déstructuré, c'est à dire l'absence de construction syntaxique de haut-niveau pour représenter les boucles, a nécessité le développement d'heuristiques de parcours de l'arbre d'exécution du programme. Ensuite, la modélisation de l'héritage dans un modèle à contraintes dérivé de l'analyse du Bytecode, a nécessité d'adapter les structures de données que nous avions développées pour le langage C. Il est à noter toutefois que ce point est moins dur à résoudre dans le cas du Bytecode Java que dans le cas de code binaire exécutable pour lequel les types sont absents et qui contient des instructions de saut dynamique[1] [Bardin 08]. Enfin, le polymorphisme par invocation de méthodes virtuelles est un problème ardu car il nécessite la mise au point d'un mécanisme capable de résoudre dynamiquement l'invocation de méthode, c'est à dire lors de la résolution du système de contraintes. Nous avons développé un tel mécanisme en nous appuyant sur des variables de type et des contraintes spécifiques sur les types.

[1] "Dynamic jumps"

F. Charreteur and A. Gotlieb. *Constraint-based test input generation for java bytecode*. **In Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10), San Jose, CA, USA, Nov. 2010.**

Constraint-based test input generation for java bytecode

Florence Charreteur
Universit de Rennes 1
Rennes, France
Florence.Charreteur@irisa.fr

Arnaud Gotlieb
INRIA Rennes Bretagne Atlantique
Rennes, France
Arnaud.Gotlieb@irisa.fr

***Abstract*—In this paper, we introduce a constraint-based reasoning approach to automatically generate test input for Java bytecode programs. Our goal-oriented method aims at building an input state of the Java Virtual Machine (JVM) that can drive program execution towards a given location within the bytecode. An innovative aspect of the method is the definition of a constraint model for each bytecode that allows backward exploration of the bytecode program, and permits to solve complex constraints over the memory shape (e.g., `p == p.next` enforces the creation of a cyclic data structure referenced by `p`). We implemented this constraint-based approach in a prototype tool called JAUT, that can generate input states for programs written in a subset of JVM including integers and references, dynamic-allocated structures, objects inheritance and polymorphism by virtual method call, conditional and backward jumps. Experimental results show that JAUT generate test input for executing locations not reached by other state-of-the-art code-based test input generators such as jCUTE, JTEST and Pex.**

I. Introduction

As integrated within current development environments, automatic test input generation is a promising approach for reducing the cost of software unit testing. In particular, having an approach able to generate and check 100% code coverage of a unit under test is highly desirable for increasing our confidence in the program correctness. However, current realistic solutions for this problem do not achieve complete coverage of usual structural criteria such as all_statements or all_branches. In fact, modern automatic test input generators adequately sacrifice completeness for efficiency. They are able to generate test inputs for programs containing hundreds of thousands lines of code in a couple of minutes but they fail to generate complete statement coverage on simple Java methods that contain cyclic data structures, multi-level dereference aliasing problems, inter-dependent loop statements or non-linear decisions.

Although the underlying reachability problem is undecidable in the general case, it seems that there is room for code-based test input generators applying more costly methods and having the goal of completeness. Constraint-based testing, as introduced by Offutt in 1991 [1], combined symbolic execution and dynamic constraint solving [2] to generate test inputs able to execute specific paths in the code. It was then adapted for C programs and refined with standard constraint programming techniques to target specific locations in the code regardless of the particular path executed [3], [4]. Constraint-based testing was extended to the exhaustive generation of structured inputs for Java programs [5], [6] and the constraint solving of structurally complex constraints [7]. An important work in the area is also the symbolic execution approach of Java PathFinder [8] where "lazy initialization" was proposed [9].

Recent code-based test input generators such as DART [10], PathCrawler [11], CUTE and jCUTE [12], [13] or Pex [14] are based on *dynamic symbolic execution*. They dynamically select a feasible path by picking up a test input and by observing which instructions are executed ; then, they report path conditions by symbolically evaluating the instructions along the path. Finally, by negating the last decision of path conditions and submitting the corresponding system to a constraint or an SMT[1] solver, they try to infer another test input covering a distinct path. When such a test input is found, path coverage of the program is necessarily increased. Whenever path conditions are unsatisfiable, the process backtracks and selects another path to execute. Dynamic symbolic execution performs *forward exploration* as it visits the paths from the entry to the exit and this works fine for quickly exploring a few paths. However, when the goal is to complement an existing test set in order to increase code or branch coverage, this approach is less adapted as forward exploration can be trapped in large subspaces of the path search space. In this paper, we introduce a new constraint-based approach to generate automatically test inputs for Java bytecode programs. Our approach generates a test input under the form of a concrete state of a JVM memory, that can drive the program execution towards a selected bytecode instruction. In the context of unit testing, our approach is made to improve the statements coverage of each method under test thanks to a context-free test case generation. This paper contains two contributions:

1) unlike other approaches [10], [12], [11], [15], [14], our framework explores program paths from the target location towards the program entry point. *Backward exploration* at the bytecode level is not trivial as it requires the development of a kind of inverse reasoning on each bytecode instruction, that is neither available in the SUN's JVM specifications nor in the literature. For that, we developed constraint reasoning on partial abstract memory states. For backward exploration, we also developed a depth-first strategy which takes advantage of early detection of infeasible parts of paths to prune the search space of feasible paths ;
2) a new constraint-based model of the JVM is defined with the notion of *constrained memory variable*. This notion captures abstract memory states and permits to implement deductive rules for a meaningful subset of Java bytecode, including those dealing with objects, inheritance, polymorphism and dynamic memory management. In this model, each bytecode expresses a relation between two *constrained memory variables*, which contain unknown or only partially known variables associated to registers, operands stack or heap. Using constraint propagation and an existing finite domain constraint solver, this model allows decisions that involve cyclic data structures and reference aliasing problems such as `if( p == p.next )` to be effectively solved.

[1]Satisfiability Modulo Theories.

We implemented our approach in a tool called JAUT (Java Automatic Unit Testing) that can generate input memory states for reaching specific locations within Java bytecode programs. Experimental results on small-sized benchmark programs, including the `TreeMap` Java library of red-black trees, show that JAUT[2] can complement an existing test set with test inputs for locations not covered by other code-based test input generators such as jCUTE[12], [13], JTEST[16] and Pex[14].

The rest of the paper is organized as follows: section 2 introduces our method on a simple example, section 3 presents the memory model we defined to generate test inputs for the JVM, section 4 explains constraint generation and constraint solving for a few bytecodes, section 5 presents the test input generation process which performs backward search across Java bytecode programs, section 6 is dedicated to our experimental validation and discussion of related work, while section 7 concludes and draws perspectives to this work.

II. Detailed example

```
class Coord {
 int x,y;
 Coord(int cx, int cy){ x = cx; y = cy; }

 public Coord moveY(Chrono chrono,int speed) {
  if(chrono.time <= 0 || speed <= 0)
      return this;
  int ytemp = y + chrono.time * speed;
  chrono.time = 0;
  if( ytemp > 65536 )
   {return new Coord(x,65536);}    //instruction i
  else return new Coord(x,ytemp); }

 class Chrono {
 int time;
 ...}
```

Figure 1. Example in Java source code

Consider the Java program of Fig.1 that implements the class `Coord` standing for the Cartesian coordinates in a 2D-space. We selected this simple example just to illustrate the memory model of our constraint-based test input generation approach. Let our test objective be the generation of an input JVM state for method `moveY` that reaches the bytecode corresponding to the instruction i in the source code. Note that we do not pay attention to how the methods of class `Coord` can build such an input state that may be "invalid" if it cannot be reached from other method calls. Object-oriented languages offer ways to directly manipulate the object receiver state without using constructors and accessors (e.g., using *Java reflection* with `setAccessible` or *bytecode translation*) and nothing can guarantee, without additional information, that a given method will not be called from an "invalid input state". Hence testing a program with states that may be invalid is equally important.

The bytecode program shown in Fig.2 corresponds to method `moveY` where bytecode 51 corresponds to the selected test objective.

[2]JAUT and all our experiments are freely accessible at http://www.irisa.fr/lande/gotlieb/resources/jaut.html .

```
public Coord moveY(Chrono, int);
Code: Stack=4, Locals=4, Args_size=3
 0:  aload_1      //push the ref. from the register 1 on the stack
 1:  getfield #4  //update the top of the stack Chrono.time
 4:  ifle 11      //if the top of stack ≤ 0, jump to 11
 7:  iload_2      //push the integer in the register 2 on the stack
 8:  ifgt 13      //if the top of the stack ¿ 0, jump to 13
 11: aload_0
 12: areturn      //return the top of stack
 13: aload_0
 14: getfield #3  //update the top of the stack with the field y
 17: aload_1
 18: getfield #4  //update the top of the stack with Chrono.time
 21: iload_2
 22: imul         //update the top of the stack with the product
                    of the two elements on the top of the stack
 23: iadd
 24: istore_3     //store the top of the stack in the register 3
 25: aload_1
 26: iconst_0     //push the constant 0 on the stack
 27: putfield #4  //update the field Chrono.time
 30: aload_3
 31: ldc #5       //push the integer constant 65536 on the stack
 33: if_icmple 52 //if the second element of the stack is less or
                    equal to the top, jump to 52
 36: ldc #5
 38: istore_3
 39: new #6       //allocate dynamic. mem. for a Coord object
 42: dup          //duplicate the top of stack
 43: iload_3
 44: aload_0
 45: getfield #2  //update the top of the stack with the attr. x
 48: invokespecial #7 //invoke the constructor Coord(int,int)
 51: areturn      //return the top of stack
 ...
```

Figure 2. Example in Bytecodes

Our memory model is based on the notion of constrained memory variables (CMV) which capture JVM states. It contains abstract information on the registers, the operand stack and the heap of the JVM, which can be used to fill in a test script for the method under test.

Our prototype tool JAUT manipulates CMVs and can output, at any moment of the constraint solving process, an excerpt of the CMV associated to `moveY` input JVM state:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[]>,
     speed -< intFD,   [-268435456..268435455]>,
hin:
   0::([Coord],[x-<intFD,[-268435456..268435455]>,
               y-<intFD,[-268435456..268435455]>])
```

Before launching the constraint solving process, the CMV containing the input parameters `lin` and the heap `hin` is automatically generated. `lin` contains two references (`this` and `chrono`) and an integer (`speed`). The object referenced by `this` has already been created on the heap (noted `0::`), as invoking method `moveY` requires a `Coord` object being created. `this` refers to this object as its domain contains a single address (`dom=[0]`). On the contrary, `chrono` does not reference any object for the moment as it can still be null or points to any object of the heap (`dom=all`). `ndom` denotes a set of impossible adresses for a heap reference. Note that references ($0, 1$ and so on) do not reflect any physical counterpart, as our model is abstract and not designed for bytecode execution. Integer variables such as `speed`, `this.x` and `this.y`, that are 32-bits integers in the

Java bytecode program, currently have type `intFD`, with domain[3] $[-268435456..268435455]$.

The constraint generation and solving process of our test input generation method aims at refining this input CMV for method `moveY` by finding values for each variable. We illustrate this process below by showing how this input CMV is successively refined to satisfy the test objective.

Our constraint solving examines each bytecode one by one in a backward fashion. On this program, there is no choice point on the backward exploration as there is only a single path going towards the program entry point. Constraint solving operates with two interleaved processes: *constraint propagation* which uses each constraint to prune the domain of CMV of their inconstistent values and *labeling*, which enumerates the possible values of CMV and launches constraint propagation until no more deduction is possible. Those processes are not new and form the basis of finite domain constraint solving [17]. Let us just recall that all the constraints are added to a constraint propagation queue that iteratively manages each constraint one by one, and each constraint is used to prune the variation domain of its variables. At a given step of the constraint solving process, the following input CMV is produced:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[null,0]>,
     speed -< intFD,   [1..268435455]>,
hin:
  0::([Coord],[x-<intFD,[-268435456..268435455]>,
              y-<intFD,[-268435456..268435455]>])
```

The `ndom` set of reference `chrono` has increased to `[null, 0]`, meaning that `chrono` can neither be null nor points to the object pointed by `this`. These deductions come from the fact that 1) reaching location 51 in the bytecode program requires a non-null value for `chrono` as it is dereferenced several times and 2) `chrono` has to point to a `Chrono` object and not a `Coord` object. The domain of `speed` has been pruned during initial constraint propagation as constraint $speed > 0$ was considered.

Next steps include the creation of object `Chrono` and the start of the *labeling* process for producing a completely instantiated CMV, suitable for test script generation. Suppose that the labeling process has instantiated `speed` to 363 and `this.y` to 5206 using a random labeling heuristic, then we get the following refined input CMV:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     speed -< intFD,   [363]>,
hin:
  0::([Coord],[x-<intFD,[-268435456..268435455]>,
              y-<intFD,[5206]> ]),
  1::([Chrono],[time-<intFD,[167..739477]> ])
```

An object of class `Chrono` has been created and is referenced by `chrono` which has now value 1. Accordingly, the domain of `chrono.time` has been pruned to $[167..739477]$ as the arithmetic constraints $ytemp = this.y + chrono.time * speed, ytemp > 65536$ were considered by the underlying finite domain constraint solver.

However, the input CMV has still uninstantiated variables such as `this.x` and `chrono.time`. Therefore, the labeling process enumerates values for these variables.

So, we get:

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     speed -< intFD,   [363]> ,
hin:
 0::([Coord], [x-<intFD,[254]>,y-<intFD,[5206]>]),
 1::([Chrono],[time-<intFD,[167]>])
```

This CMV characterizes an input JVM state that satisfies the test objective of reaching bytecode numbered 51 in method `moveY`. Submitting a test script derived from this state shows that the fault on the converted parameters of `new Coord(ytemp, x)` can be detected.

III. Memory model

Java virtual machine states represent runtime data storage locations such as registers[4], operand stacks and heap data. This section details the representation of data types and data storage locations in our memory model. Note that our framework handles a meaningful subset of Java bytecodes, including integers and references, dynamic allocation and heap-allocated structures, objects inheritance and polymorphism, static and dynamic method invocations, conditional jumps and backward jumps. Arrays accesses and updates are only partially supported and floating-point computations, exceptions, native methods and multithreading are currently left apart.

A. Constraint memory variable

In our memory model, *constrained memory variables* (CMV) are used to represent JVM states. A CMV contains data storage locations where data can be represented by variables along with a domain. Formally, a CMV M is a tuple (F, S, H) where F denotes the set of registers, S denotes the operand stack and H denotes the heap. Note that several distinct CMV M can be created at the same instruction number when loops are present in the bytecode. Each Java bytecode will then be seen as a relation among two CMVs: the CMV M_j before activation of bytecode and the CMV M_k after its activation and before the activation of the following bytecode in the considered sequance of instructions. The tuple (F, S, H) contains variables and domains that are described in the rest of the section.

B. Integer and reference variables

Finite domain variables model integer and reference variables of the program. Their default variation domain depends on the size of their precise type. For instance, the domain $-2^{31}..2^{31}-1$ is associated to an `int` variable. Other integer types are treated accordingly. The default domain of a reference is the *all* symbolic value. This means that the reference can point to every object of the heap. When the solving process prunes the domain of the reference variable, then the domain is composed of a set of integer values, which represent all the heap addresses the reference can point to. Note that the *null* value can also be part of the domain.

C. Objects

Each object of the heap is modeled by a pair of elements. The first one, called *type variable*, represents the class of the object, the domain of which is a set of possible classes. This allows to properly handle inheritance and polymorphism. The second element is a mapping associating an integer or a reference variable to each attribute, which corresponds to the value of the attribute. Note that when the domain of the *type variable* contains more than one class, all the possible attributes have to be in the mapping.

For example, if a program defines two classes `A` and `B` where `B` inherits from `A`, and `A` defines attribute `t1` and `B` defines attribute `t2`, then

```
([A,B],[t1-<intFD,[2..5]>,t2-<intFD,[15..17]>])
```

represents an object of class either `A` or `B`, with attribute `t1` necessarily in $2..5$. If the object happens to be of class `B` during the solving process, then its `t2` attribute will belong to $15..17$.

D. Registers and operand stack

In a JVM state, registers are used to store the parameters and the local variables of a method. When the method is dynamic (as opposed to `static` methods), the first register contains the reference to the object (`this`) that calls the method. The operand stack is used to perform the calculations of the method. In a CMV, we use two sequences of variables to represent registers and operand stack. As registers are numbered, the first sequence is a convenient way of implementing an indexed array. On the contrary, the sequence used for representing operand stack is accessed with stack operations. Its first element is considered as its top.

E. Heap

In a CMV, the JVM's heap corresponds to a mapping from a set of addresses to a set of objects, possibly stored at these addresses. We associated a unique integer to each address without taking into consideration the actual physical addresses in the JVM. The domain of a variable H that models the heap is composed of a set of pairs, representing the mapping, and a status. 1) The set of pairs *(address,object)*, denoted by C_H, contains the objects necessarily present in the heap. During the solving process, new pairs can be added to C_H when a dynamic memory allocation bytecode is encountered (i.e., `new`). As our model does not model garbage collection, C_H can only be enriched, reducing the possible states of the heap. 2) The status is either $closed$ or $unclosed$. A $closed$ status of the heap denotes a set of addresses entirely known, even if some objects can still be unknown or only partially known through their domain. On the contrary, an $unclosed$ status denotes states where memory allocation can still enrich the CMV. During the constraint solving process, status can only move from $unclosed$ to $closed$. This notion is introduced to tackle cases where the input memory shape is unknown and we will have to explore the input space during labelling to find it.

IV. Constraint generation and solving

In our framework, constraint generation and solving are performed altogether. This has similarities with the on-the-fly generation of paths of [11] and [14] where path conditions are incrementally tested for satisfiability. We first detail the specification of test objective (Sec. IV-A), and then we present the constraint generation and solving process (Sec. IV-B). We illustrate the constraint generation on an example (Sec. IV-C) and we explain the deduction capabilities of the constraints we have defined to model dynamic memory management (Sec. IV-D).

A. Test objective

Test input generation aims at finding an input CMV that satisfies a given test objective in a Java bytecode method. In our framework, test objectives are specified with the *bytecode instruction number* that corresponds to a bytecode program location. In general, for a given test objective, there are many feasible or infeasible paths that can reach the target locations. Our goal is just to find one input CMV that will drive the computation towards the selected location, whatever is the feasible path executed. Note that such a test objective potentially specifies an infinite set of dynamic locations when the bytecode instruction number is located within loops.

B. Constraint generation from the bytecode

Initially, the input CMV of the method is an unconstrained variable and the constraint generation process will accumulate and solve the constraints, for the current selected path, from the test objective to the program entry point. These constraints capture the pah condition that must be satisfied to follow the current selected path and are used to prune the possible values of the input CMV. Each bytecode can potentially constrain the input CMV variable or intermediate CMV variables, throughout its behavior on the registers, operand stack or heap variables. Based on the semantics of each bytecode as defined in the SUN's specification, we build constraints that implement deductive rules on the CMVs.

Arithmetic bytecodes such as `iadd, ladd, isub, imul, idiv, irem, ineg,...` and comparison bytecodes `if_icmp, if_acmp, lcmp, ...`, act directly on logical variables associated with integers or references, and the elements of the operand stack. They generate arithmetical constraints on the finite domain constraint solver, depending on the type of the operands and the operator. For example, `iadd` generates a relation $Vtemp = Va + Vb$ over two CMVs $M1$ and $M2$, where $Vtemp$ is a fresh finite domain variable associated to the top of the operand stack of $M2$, while Va and Vb are the two finite domain variable associated with the integers on the top of the stack of $M1$. Other arithmetic bytecodes generate similar constraints according to the considered integer type (`int` or `long`) and operator. Comparison bytecodes are handled in a similar way by considering the arithmetic constraint extracted from the condition or from the negation of the condition. This will be made clearer in section V.

Bytecodes for simple constant pool accesses (`ldc, bipush, ...`) or register accesses `iload, aload, iload_<n>, aload_<n>, ...` or register updates `istore, astore, istore_<n>, astore_<n>, ...`, generate equality constraints between the top of the operand stack and registers, depending on the type of the register variable. Equality on reference variables (when `aload` or `astore` is considered) generates a finite domain equality, as well.

To deal with bytecodes for dynamic memory management `new, newarray, ...` and accesses and updates of object fields `getfield, putfield, ...`, special constraints

[3]There is a current 28-bits limitation for integers in our implementation, due to the use of the SICStus prolog `clpfd` library, but this is not a restriction of the model.

[4]a.k.a. local variables in the SUN JVM specifications.

have to be built. Indeed, these operations maintain complex relations between the CMVs. They can constrain the shapes of heap-allocated data structures and the contents of registers and operand stack. We detail the relations we built for only three of them, as the other can easily be deduced from these ones: $new(class, H0, H1, A)$ maintains the relation between $H0$ the heap of the CMV before execution and $H1$ the CMV after execution, and A a fresh address for the newly created object of type $class$. The relation says that $H1$ is the updated heap $H0$ where reference A points to a newly allocated object.
$getfield(A, Id, H, Val)$ maintains the relation between the heap H, from which an access to an attribute Id of the object designated by reference A is performed and Val a finite domain variable. When reference A has for domain a set of possible references, special deductions can be performed on the possible values for Val. Conversely, using the possible values of Val, special deductions on reference A can be performed. In addition, information on the domains of variables in H allows for deductions on the domains of A and Val. This permits the implementation of powerful forward and backward constraint reasoning.
$putfield(A, Id, Val, H0, H1)$ is more complex as it maintains a relation between $H0$ and $H1$ the heaps of both CMVs, reference A and Val. $H1$ is similar to $H0$ except for object referenced by A where the field Id has been modified to value Val. Possible deductions with this relation are illustrated below.
Bytecodes for arrays manipulation `baload, iaload, iastore, ...` and method invocations `invokespecial, invokevirtual, invokestatic` are taken into account but they are not described here as we considered they were outside the scope of the paper.

C. Example

From the `moveY` method of Fig.2 and the test objective which consists to reach bytecode numbered 51, we get the following constraint system (omitting some details about the calls to constructors). For this example, M_i denotes the memory state before the bytecode instruction number i.

$$
\begin{array}{l}
\{\ F1 = This.Chrono.Speed.Ytemp,\ This \neq null \\
\quad M_{51} = (F1, S1, H1), \\
\quad M_{48} = (F1, X.Y.A1.S1, H2), \\
\qquad invokespecial(Coord, init, [A1, Y, X], H2, H1), \\
\quad M_{45} = (F1, A2.Y.A1.S1, H2), A2 \neq null, getfield(A2, x, H2, X) \\
\quad M_{44} = (F1, Y.A1.S1, H2),\ A2 = this, \\
\quad M_{43} = (F1, A1.S1, H2),\ Y = Ytemp, \\
\quad M_{42} = (F1, A1.S2, H2),\ S1 = A1.S2, \\
\quad M_{39} = (F1, S2, H3),\ A \neq null,\ new(Coord, H3, H2, A1), \\
\quad M_{38} = (F2, Ytemp.S2, H3),\ F2 = This.Chrono.Speed.Ytemp2, \\
\quad M_{36} = (F2, S2, H3),\ Ytemp = 65536, \\
\quad M_{33} = (F2, Val1.Val2.S2, H3),\ Val2 > Val1, \\
\quad M_{31} = (F2, Val2.S2, H3),\ Val1 = 65636, \\
\quad M_{30} = (F2, S2, H3),\ Val2 = Ytemp2, \\
\quad M_{27} = (F2, Val3.A3.S2, H4),\ A3 \neq null, \\
\qquad putfield(A3, time, Val3, H4, H3), \\
\quad M_{26} = (F2, A3.S2, H4),\ Val3 = 0, \\
\quad M_{25} = (F2, S2, H4),\ A3 = Chrono, \\
\quad M_{24} = (F3, Ytemp2.S2, H4),\ F3 = This.Chrono.Speed.Ytemp3, \\
\quad M_{23} = (F3, Val4.Val5.S2, H4),\ Ytemp2 = Val4 + Val5, \\
\quad M_{22} = (F3, Val6.Val7.Val5.S2, H4),\ Val4 = Val6 * Val7, \\
\quad M_{21} = (F3, Val7.Val5.S2, H4),\ Val6 = Speed, \\
\quad M_{18} = (F3, A4.Val5.S2, H4),\ A4 \neq null,\ getfield(A4, time, H4, Val7), \\
\quad M_{17} = (F3, Val5.S2, H4),\ A4 = Chrono, \\
\quad M_{14} = (F3, A5.S2, H4),\ A5 \neq null,\ getfield(A5, y, H4, Val5), \\
\quad M_{13} = (F3, S2, H4),\ A5 = This, \\
\quad M_{8} = (F3, Val8.S2, H4),\ Val8 > 0, \\
\quad M_{7} = (F3, S2, H4),\ Val8 = Speed, \\
\quad M_{4} = (F3, Val9.S2, H4),\ Val9 > 0, \\
\quad M_{1} = (F3, A6.S2, H4),\ A6 \neq null,\ getfield(A6, time, H4, Val9), \\
\quad M_{0} = (F3, S2, H4),\ A6 = Chrono, S2 = \epsilon\}
\end{array}
$$

ϵ is the empty sequence, while $v.s$ denotes the stack s where v is pushed. The elements of the sequence $F2$ represent the parameters and the local variables at the bytecode instruction 51. Instruction 48 calls a constructor and links both CMVs M_{48} and M_{51}. The elements on the top of the stack before instruction are the parameters Y and X as well as the reference $A1$ to the object that calls the method. The heaps $H2$ and $H1$ are linked by the constructor call effect, represented here by a relation $invokespecial$.
The top of the stack before the instruction $getfield$ 45 is a reference $A2$, while the top of stack X after the instruction is the value of the attribute x of the object referenced by $A3$ in the heap $H1$. The relation is maintained by a constraint $getfield(A3, x, H2, X)$. The instruction 39 allocates memory to store a new object of type $Coord$. The relation $new(Coord, H3, H2, A1)$ states that the heap $H2$ contains one added object of type $Coord$. $A1$ is pushed on top of the stack after the instruction. As the only instruction that permits to reach 51 after the conditional instruction 33 is 36, then the top of the stack $A.B$ before the instruction 33 constrains B to be greater than A.
The top of the stack before the instruction 27 contains the value $Val3$ and the reference $A4$. The instruction $putfield$ updates the value of the attribute $time$ with the value $Val3$, for the object referenced by $A3$. $H4$ is the heap before the instruction while $H3$ is the heap after instruction and the constraint $putfield(A3, time, Val3, H4, H3)$ maintains this relation. Finally, the values of the registers $F3$, and the heap $H4$, of the memory M_0, describe a possible test input to reach our test objective 51.

D. Possible deductions with operator putfield

In order to illustrate the behavior of a complex constraint, we show the possible deductions by $putfield$ on a simple example. Consider the heap $H0 = \{(1, (a, [t1 \mapsto V1, t2 \mapsto V2])), (2, (a, [t1 \mapsto V3, t2 \mapsto V4])), (3, (b, [t3 \mapsto V5]))\}$ and the heap $H1 = \{(1, (a, [t1 \mapsto V6, t2 \mapsto V7])),$ $(2, (a, [t1 \mapsto V8, t2 \mapsto V9]), (3, (b, [t3 \mapsto V10])$, which both contain two objects of class a and one object of class b, and the relation $putfield(A, t1, Val, H0, H1)$ where $t1$ is the first of the two attributes of the class a. Class b, which does not inherit from a, has only a single attribute. Suppose that $dom(V1) = [0..10]$, $dom(V3) = [1..3]$, $dom(V8) = 15..2^{31} - 1$ while other domains are unconstrained, let $dom(A) = \{all\}$ and $dom(Val) = [10..40]$, and suppose that status of $H0$ is *unclosed* and status of $H1$ is *closed*. Using the relation $putfield(A, t1, Val, H0, H1)$, several deductions can be performed. 1) As the relation operates on attribute $t1$, one deduces that $dom(A) = all - \{3\}$ as object 3 has only attribute $t3$ and its class b does not inherit from a. Consequently, the following equalities can be added $V2 = V7, V4 = V9$ and $V5 = V10$ as the relation does not modify variables associated with the attributes $t2$ and $t3$. 2)Considering variables $V3$ and $V8$, we see that $dom(V3) \cap dom(V8) = \emptyset$ then one deduces that A refers necessarily to object 2 in the heaps $H0$ and $H1$: $A = 2$. Consequently, $V1 = V6$ is added as the object at the address 1 is not modified and $V8 = Val$ leading to $dom(V8) = dom(Val) = [15..40]$. 3) Finally, the *closed* status of $H1$ is propagated to $H0$ as $putfield$ does not add any new object on the heap. So, all the addresses of both heaps are known.

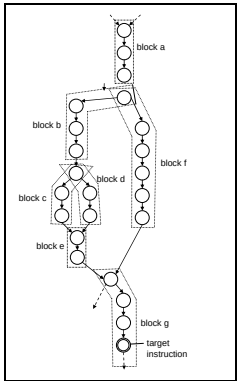


Figure 3. Constraint blocks

V. Test input generation

A. Backward search

Backward search relies on constraint block accumulation and solving. A *constraint block* in a bytecode program is a set of instructions that is submitted to constraint propagation. The test objective specifies a bytecode to reach, which corresponds to a specific constraint block. So, starting from this block, backward search incrementally accumulates other constraint blocks by trying to find a path towards the program entry point. It is worth noticing that constraint consistency is tested on the fly, each time a new constraint block is added to the constraint system. This permits to quickly detect infeasible parts of program paths. These subpaths should not be confounded with path prefix as they do not begin by the method entry point. When a constraint block has several parent blocks in the control-flow graph, then a choice point is created. Each choice is then explored in a depth-first search until one of them gets to the program entry point. If a choice is shown as being incoherent with the rest of the constraint system, then the process backtracks to the first ancestor choice point and takes an alternative. The strategy gives priorities to shortest intraprocedural subpaths towards the program entry points (i.e., in presence of loops, targeting exit loop constraint blocks). Note however that a parameter has to be set to prevent the process from iterating without terminating in presence of loops.

Let us illustrate why backward search can be interesting w.r.t. forward exploration on the simple example of Fig. 3, where the test objective is to reach constraint block g. Suppose also that block e and block g contain contradictory conditions. For example, e contains assignment `i = 1` while condition to go from block e to block g is `i > 2`. Starting from block g, backward search will first explore subpath $g - e$ by positioning a choice point. As constraint consistency is tested on the fly, the process fails and backtracks to the choice point without exploring the other backward paths starting by $g - e$. Subpath $g - f - a - ...$ is then considered without failing. Constraint inconsistency detection is possible as our framework can reason both in a forward and backward manner, thanks to the use of contraint programming to express relations between CMV. Moreover, our framework implements a precise memory model able to deal with partially known memory states. On this example, forward exploration such as implemented in several dynamic symbolic execution tools would have first considered $a - b - c - e - g$ and failed. Then, backtracking at point b, depth-first forward exploration would have considered $a - b - d - e - g$ and failed again before finding $a - f - g$. To be fair, similar poor behavior can be found for backward exploration as well just by considering incoherent subpaths near the program entry point. We are not saying that backward exploration is better than forward, we just say that both approaches complement each other. Our backward search approach is useful to complement an existing test set by looking at not-covered locations.

B. Test input labeling

When a set of consistent constraint blocks has been found to flow backward from the test objective to the program entry point, every variable of the input CMV has not necessarily been instantiated. A labeling step on the input CMV has then to be launched in order to complement the CMV. The process is recursive and tries to select the best option at each step. It starts by labeling the input formal parameters of the method under test. For references, it first tries $null$, then it tries the addresses of objects in the heap that have compatible type and finally, upon failure, it creates a new object that has a compatible type. Each time a value is assigned to a variable from its domain, the constraints that operate on this variable are awaked and constraint propagation is relaunched. This process can efficiently prune the domain of other variables. When all the input parameter values are fixed, the values of the object attributes have to be labeled in order to build a complete CMV. The process labels first the object attributes that are reachable from the parameters with a single dereferencing level. If the corresponding input state satisfies the test objective, then we get a solution of the problem. On the contrary, upon failure, the process backtracks and the dereferencing level is increased until a given bound, defined by the user. The status is also labeled to the "closed" value in order to indicate that no more object can be added to the input memory state. When the labeling process is finished, then either a complete input CMV is found that reaches the test objective or a failure is reported. Failure may be provoked by several causes, including unreachable test objective, bounds on backward search or dereferencing level, or timeout. Hence, the method is incomplete but note that it can find input cyclic data structures when necessary.

VI. Experimental Validation

A. Implementation

Our prototype tool JAUT (Java Automatic Unit Tesing) takes as inputs a bytecode program given under textual form, obtained with SUN's command `javap` which decompiles the binary bytecode, and a test objective composed of the method name and bytecode instruction number. It reports as output an input CMV, excerpts of which have been presented in Sec. 2 of the paper. JAUT includes 1) a bytecode analysis module that generates a prolog-based internal

structure that can be efficiently explored with backtracks and unification ; 2) a backward search module which is parameterized by: a bound on the number of times certain branches are executed, a bound on the length of paths to be explored, a bound on the dereferencing level and a timeout ; 3) a constraint solver which implements constraint generation and deduction rules for the constraints associated with bytecodes. The constraint solver implements its own constraint propagation queue and its own memory labeling strategy, but it calls the SICStus Prolog `clpfd` library for solving arithmetical constraints.

JAUT handles a meaningful subset of bytecodes, as discussed earlier, but as a research prototype tool, it handles only about a third of the one hundred or so bytecodes of the SUN's JVM specification. Bytecodes that were left apart include bytecodes for integer conversions and low-level shifting, bytecodes for floating-point computations. Note that, for each bytecode, a specific constraint model was developed and deduction rules that captures the operational semantics of the JVM were implemented.

B. Experiments

All the results were computed on a standard single-core machine: an Intel Pentium, 2.16GHZ machine running Windows XP with 2.0GB of RAM. Our experiments aimed at evaluating the capabilities of JAUT to complement an existing test set, obtained by another forward path-exploration test data generator. We selected three such white-box test data generation tools: jCUTE, JTEST version 8.0 and Pex version 0.15.40714.1. Pex is dedicated to .NET but the programs considered in our experiments can easily be compiled to .NET bytecodes[5] Other available test data generators are discussed in the related work section. The goal of the experimental settings was to evaluate the capabilities of JAUT to cover instructions not covered by the three test input generators jCUTE, JTEST and Pex.

Simple programs are used in our experiments: versions of the classical `trityp` and `josephus` programs in bytecodes, methods of the `DoublyLinkedList` and `TreeMap` Java classes. `Trityp` has many infeasible paths while `josephus` [18] manages a cyclic dynamic data structure. We also considered a modified version of this program, called `josephus/m`, where the hard-to-reach decision `ndeEnd.key==41 && nde.key==31` is inserted at the end of the program. The `DoublyLinkedList` class also implements dynamic data structures management and contains input object references and method calls in its code. The `TreeMap` class implements red-black trees which are cyclic structures. The source code of these programs, the source code of JAUT, as well as all the test drivers used for Pex and jCUTE can be found online at `www.irisa.fr/lande/gotlieb/resources/jaut.html`. In all the programs, private fields have been turned into public ones in order, for the three tools, to equally generate valid and invalid input states. For the depth-first backward exploration of JAUT, a bound of 150 bytecodes on the length of path and a bound of 10 for the maximum dereferecing level have been set.

In addition, we considered the following program (in bytecodes) that illustrates a problem related to forward exploration:

```
static int a=1;
```

[5]In our results, difference of the .NET runtime cost vs. JVM runtime have been considered neglectible.

```
public static int foo(int i){
      int j = 10 ;
      while (i > 1){ j++;  i--; }
      if (j > 50*a)
              return 1;  // test objective I
      return 0; }
```

Reaching the test objective I implicitely constrains the number of iterations within the loop. On this example, forward exploration will unroll the loop without taking into account the test objective. The parameter `a` can be increased to study the scaling effect of the underlying problem.

C. Results and analysis

On the `foo` program, both JTEST and Pex fail to reach the test objective I. Unlike these tools, jCUTE covers every instructions of the `foo` method, including instruction I, but it takes 10.9sec of CPU time. JAUT also generated an input CMV (corresponding to $i = 42$) that reaches I in 0.15sec of CPU time. Its backward exploration strategy is useful on this example.

We studied the behavior of both jCUTE and JAUT on the `foo` example by increasing the value of `a`. Fig. 4 shows the results of this experiment. The time required by jCUTE increases dramatically on this example as the tool requires an increasing number of trials to satisfy this hard-to-reach test objective. As jCUTE compiles and executes programs to perform dynamic symbolic execution, the CPU time increases accordingly. The JAUT approach is better on this example as constraints are directly handled in the constraint solver and backtracking is hard-coded in Prolog. For other experiments, results are shown in Fig.5 where #Bc is

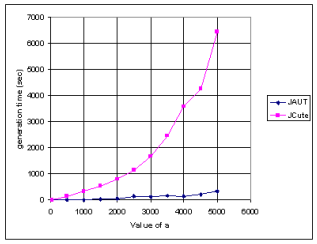


Figure 4. CPU Time to generate input for I in function of a, for method **foo**

the number of bytecode instructions, #To is the number of test objectives. We report the results of jCUTE, Pex, and JAUT to get **complete bytecode instructions coverage** and the results of JAUT to complement the test sets produced by Pex. The fourth tool JTEST reveals itself too poor on this task. In fact, its test input generation strategy is based on the analysis of constants in the program, which prevents many symbolic decisions to be covered. jCUTE and Pex have the primary goal of covering all the feasible paths of the program while JAUT is designed to reach a single instruction in the program. Hence, none of these three tools is optimized to efficiently get complete bytecode instruction coverage. However, in all the cases, there are claims on their ability to reach instruction coverage. For JAUT, instruction coverage can be automatically reached by launching incrementally requests on not-covered instructions in a script. For the three tools, we report coverage percentage, which we have computed ourselves on a common basis. Indeed, Pex reports dynamic bytecode coverage which does not necessarily correspond to bytecode instruction coverage. So, we computed the coverage by looking at the covered portions of code, rather than taking into account the numbers provided by the tools. We selected two search modes for jCUTE: the former corresponds to depth-first search on the execution tree (both columns noted *jCUTE*) while the second corresponds to random path exploration (both columns noted *jCUTEr*). We measured the CPU time required by JAUT to complement the test set generated by Pex (column *compl. JAUT*) when it was incomplete. When this measure was unnecessary, **n/a** was reported.

For the `trityp` method, experiments show that, unlike jCUTE in both versions, Pex and JAUT succeed to get complete coverage. However, the CPU time required to get this result for JAUT is very long (almost one hour). Indeed, SICStus Prolog needs a long time to demonstrate the insatisfiability of some path condition, as $i + j > k, j + k > i$, $i + j > k$, $j \neq k$, $i \neq k$, $i \neq j$, $k \neq 0$, $j \neq 0$, $i \neq 0$ where i, j, k denote three 32-bit integers, because it requires long-term labeling. Note that another approach would be to use dedicated SMT solver such as Z3 [19] in JAUT to avoid these problems.

Program `josephus` in both versions is hard to cover. The program computes first a dynamic cyclic chain in function of input parameters and then it eliminates one by one the elements by cycling around the chain. Satisfying all the test objectives require unrolling each of the loops 40 times which is not easy to deduce from code analysis only. On this example, Pex obtains very good results by generating 19 test cases while other tools, including JAUT, fail to get complete coverage. Our analysis of the JAUT failure indicates that our depth-first backward search is trapped by the second loop and that it should be refined. This leaves room for improvments.

For the `DoublyLinkedList` class, the three tools succeed to generate complete test sets but JAUT obtains the best CPU time results. Upon analysis of the results for the `add` method, we saw that backward exploration is useful on this example, as forward search requires many paths to be explored. To be fair, it is worth noticing that Pex also tries to cover paths of the called methods while JAUT only considers test objectives within the method under test. In addition jCUTE and Pex also generate test scripts during test input generation, while JAUT do not perform similar generation. On short periods of time, this can have an non-neglectible impact on the results. We did not implement this feature in JAUT as our primary goal was only to demonstrate the interest of fine-grain memory model and backward exploration in constraint-based test input generation.

For the `TreeMap` class, both coverage and CPU time results with JAUT outperform the results of other tools, including Pex. On the `rotateLeft` method, both versions of jCUTE obtained only 20% of bytecode coverage while Pex obtained 88.9%. In fact, covering all the instructions of the method requires explicit solving of the decision `if(p == p.left.parent)` in a certain context, which corresponds to a complex pointer aliasing relation. Note that the time required by JAUT to complement the test set of Pex when covering this decision is very low: 0.05sec. This shows that such decision does not require heavy constraint solving effort. The results obtained for method fixAfterInsertion are the most interesting. Both modes of jCUTE cover only about 20% of the method while Pex covers about 45%. The CPU times required to get these results is rather long for jCUTEr and Pex (resp. 28.30 and 120.00sec). On the contrary, JAUT covers 100% of the method in 22.18sec. When used to complement the Pex test set, it takes 15.49sec, showing that not-covered bytecode instructions are indeed hard to reach. So, JAUT performs well on the methods of the `TreeMap` class as it contains many hard test input generation problems. We interpret this result as a confirmation that using a precise memory model, which in particular deals with reference aliasing, is advantageous for completing an existing test set.

D. Related work

Test input generation at the Bytecode level. JAUT performs constraint-based test input generation from Java bytecode. Therefore, it is mainly related to JPF [8], jCUTE [12], [13], and Pex [14]. Unlike these three tools, JAUT implements backward exploration, meaning that it starts from a target bytecode location and incrementally discovers a feasible path towards the entry. Indeed, JPF, jCUTE and Pex are based on forward symbolic execution which consists to evaluate symbolically the instructions along a path in the same order as execution. Improvements on the search strategy of Pex have been proposed [20] to leverage some of the problems of forward exploration. Note that backward symbolic execution exists for a long time [21], but our approach extends and generalizes this idea to programs containing references on the stack and the heap, as well possibly cyclic dynamic data structures. Reasonning backward at the bytecode level requires a precise memory model to be defined. Our memory model has similarities with those of JPF [8], [9] in that it also implements a form of "lazy initialization". In our approach, decisions such `if( p.next == p )` constrains two variables to be equal and non null without instantiating them. Unlike lazy initialization which makes choices during constraint solving, JAUT reports the choice to the labeling phase. Our memory model also has similarities with the constraint-based models of CUTE [12] and Pex [14] that can deal with symbolic pointer equalities and inequalities, dynamic memory allocation and input data structures. We argued below that JAUT can complement an existing test set produced by one of these tools by applying more costly analysis for specific unreached locations within the bytecode. In fact, the constraint-based reasoning step of JAUT defines precise relations between two CMVs (constraint memory variables) for each bytecode. Of course, the scope of JAUT is currently too restricted (a meaningful subset of JVM bytecodes) to compete with the industrial development of Pex, but we believe that both approaches could complement each other in the long term.

Test input generation for binary programs. OSMOSE [22] and SAGE [23] both implement forward dynamic symbolic execution for binary programs. They handle dynamic jumps and use an untyped memory model able to deal with pointer arithmetic, type casting and multiple dereferencing levels. SAGE builds path conditions that are submitted to the SMT solver Z3 [19]. Unlike SAGE [23], JAUT is based on a constraint propagation and labeling solver. Such solvers are used with success in the context of combinatorial optimization problems as they can solve linear as well as non-linear

Methods	#Bc	#To	jCUTE (cov)	jCUTE (sec)	jCUTEr (cov)	jCUTEr (sec)	Pex (cov)	Pex (sec)	JAUT (cov)	JAUT (sec)	*Compl.JAUT* (sec)
trityp	89	24	83.3%	2.20	87.5%	30.88	100%	4.17	100%	3132.00	n/a
foo	15	3	100%	10.90	75%	18.80	66%	4.51	100%	0.17	0.15
josephus	51	3	100%	1.06	100%	1.06	100%	8.07	100%	0.36	n/a
josephus/m	60	5	70%	192.00	70%	**	100%	8.41	60%	**	n/a
Node class											
insertBefore	36	4	100%	3.02	100%	35.80	100%	4.03	100%	0.20	n/a
DoublyLinkedList											
pop	13	2	100%	2.16	100%	23.00	100%	0.59	100%	0.13	n/a
add	42	4	100%	9.19	88%	45.00	100%	6.09	100%	0.14	n/a
remove	31	4	100%	2.94	100%	1.89	100%	0.69	100%	0.16	n/a
RedBlackTree											
rotateLeft	48	5	20%	26.56	20%	25.10	88.9%	0.65	100%	0.23	0.05
deleteEntry	124	14	50%	2.12	91.6%	5.64	78.6%	56.00	100%	1.44	0.25
fixAfterDeletion	175	11	36.3%	1.11	54.5%	**	81.9%	92.00	100%	7.78	0.88
fixAfterInsertion	127	9	19%	1.30	18.7%	28.30	44.4%	120.00	100%	22.18	15.49

Figure 5. Time and coverage with jCUTE, Pex and JAUT (**: timeout of 3600sec)

problems over the reals or the integers. Variable multiplication or divisions are typical examples of non-linear constraints that are efficiently handled by these solvers. Another advantage of constraint propagation relies on its flexibility to add new user-defined constraints. For JAUT, we built several constraint operators that apply deduction rules to reason over the memory shapes. These operators captures both forward and backward exploration at the same time and propagates domain reductions without instantiating any variable or equality. However, using a SMT-solver such as Z3 [19] to solve arithmetical constraints over fixed-length integer variables would be very interesting to replace `clpfd` in JAUT. Note also that test input generation for binary programs is harder than at the bytecode level as the control flow cannot easily be recovered and variable types are unknown which makes the building of constrained input memory states more complex.

Test input generation from C and Java source code. There are many approaches of automatic test input generation from source code. In our previous works [4], [18], we built memory models for constraint-based test data generator of C programs. Our previous model handled pointers toward named locations of the memory and dynamically allocated structures but they were limited in their scope. Unlike JAUT, the model of [4] did not contain representation of the heap and then it was unable to deal with dynamic memory allocation. The model of [18] was very complex and the generation took too much time. Furthermore, in these preliminary memory models, backward exploration has not been implemented. PathCrawler [11], DART [10], CUTE [12] implement **dynamic symbolic execution** for C source code. On the contrary, JAUT implements **static backward exploration** These two approaches complement each other as dynamic symbolic execution rapidely covers many feasible paths while static backward exploration focusses on hard-to-reach instructions. One advantage of dynamic symbolic execution is that it can deal with third-party libraries or calls to native methods while static backward exploration cannot as there is no available constraint model. Another difference concerns the capabilities of dynamic symbolic exploration to use concrete value instead of symbolic ones for simplifying constraint solving. Note that the constraint reasonning model of JAUT handles complex constraints and just reports value choices to the labeling phase. EXE [15] implements global symbolic evaluation by launching an eager path exploration of the C program under test. Although this approach is appealing if implemented on a grid platform, it can reveal disastrous for programs that contain a huge number of paths. As soon as a program contains a loop, its number of paths is unbounded and even when it contains only conditionals, its number of path grows exponentially with the number of decisions in the worst case. Unlike JAUT, EXE does not use incremental constraint solving, so inconsistencies due to infeasible paths may be lately discovered and this could penalize the test input generation if implemented on a single machine. Note that EXE uses the STP SMT-solver which won several competitions and that it implements nice optimizations such as constraint caching and symbolic memory accesses tracking. However, as pointed out by the authors, its memory model does not handle double pointer dereferencing levels. The memory model of JAUT handles multiple dereferencing levels, up to a user-defined bound. Exhaustive bounded testing of Java programs as implemented in TestEra [5] and KORAT [6] is a test input generation method that exhaustively explores the input search space. The approach can generate a large number of dynamically allocated structures but, unlike JAUT, it does not target specific locations or paths in the code. In fact, these constraint-based approaches have developed complex strategies to efficiently generate test inputs [7] but they cannot solve path constraints extracted from programs.

Counter-example generation. Software model-checkers such as Save [24], Blast [25], Magic [26] or Cbmc [27] explore the paths of a bounded model of C programs in order to find a counter-example path to a temporal property. Some of them also address *statement reacheability* by generating test inputs to reach specific locations within the source code [8]. Some of them exploit *predicate abstraction* to boost the exploration in the context of CEGAR that stands for (Counter-Example Generation through Abstraction Refinement). JAUT contrasts with these model-checkers and CEGAR as it does not abstract the program and does not generate spurious counter-example paths. In particular JAUT builds a constraint model of bytecode program by capturing an error-free concrete semantics without considering a boolean abstraction of the program structure. On the one hand, this allows for precise input memory state to be built but, on the other hand, requires costly analysis to be implemented.

VII. Conclusions and further work

In this paper, we proposed a new constraint-based test input generation approach for testing Java programs at the bytecode level. We developed a logical memory model for a meaningful subset of bytecodes and proposed deductive rules to solve complex constraints such as `p == p.next`. Unlike other approaches, our prototype tool JAUT implements depth-first backward search and our experimental results indicate that this strategy complements forward search test data generation. Several short term improvements include the use of more dedicated constraint solvers to solve arithmetical constraints of the path conditions (e.g., Z3). Backward search could also be refined with an iterative bounded depth-first strategy to improve our handling of nested and sequential loop computations. Finally, our memory model could be improved by considering more fine-tuned awakening conditions. In the long term, our memory model could be completed to deal with exceptions as they just represent new control flow structures. Multi-threading also appears as an essential topic in Java programming and implementing a backward search strategy for test input generation of multi-threaded bytecode programs would certainly be beneficial to the community.

References

[1] R. DeMillo and J. Offut, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[2] J. Offut, Z. Jin, and P. J., "The dynamic domain reduction procedure for test data generation," *Software–Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[3] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'98)*, 1998, pp. 53–62.

[4] A. Gotlieb, T. Denmat, and B. Botella, "Goal-oriented test data generation for pointer programs," *Information and Soft. Technol.*, vol. 49, no. 9-10, pp. 1030–1044, Sep. 2007.

[5] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proc. of the 16th IEEE int. conf. on Automated soft. eng. (ASE'01)*, 2001, p. 22.

[6] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proc. of the int. symp. on Soft. testing and analysis (ISSTA'02)*, 2002, pp. 123–133.

[7] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient solving of structural constraints," in *Proc. of the int. symp. on Software testing and analysis (ISSTA'08)*, 2008, pp. 39–50.

[8] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation in java pathfinder," in *Proc. of ISSTA'04*, 2004.

[9] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transfer*, vol. 11, pp. 339–353, 2009.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proc. of PLDI'05*, 2005, pp. 213–223.

[11] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proc. Dependable Computing - EDCC'05*, 2005.

[12] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proc. of ESEC/FSE-13*. ACM Press, 2005, pp. 263–272.

[13] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *18th Int. Conf. on Computer Aided Verification, (CAV'06)*, ser. LNCS 4144, 2006, pp. 419–423.

[14] N. Tillmann and J. de Halleux, "Pex: White box test generation for .net," in *Proc. of the 2nd Int. Conf. on Tests and Proofs*, ser. LNCS 4966, 2008, pp. 134–153.

[15] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "Exe: automatically generating inputs of death," in *Proc. of Comp. and Communications Security (CCS'06)*, 2006, pp. 322–335.

[16] R. Grehan, "Jtest continues its trek toward code-testing supremacy," in *InfoWorld*, Oct. 2006.

[17] K. Marriott and P. Stuckey, *Programming with Constraints : An Introduction*. The MIT Press, 1998.

[18] F. Charreteur, B. Botella, and A. Gotlieb, "Modelling dynamic memory management in constraint-based testing," in *TAIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.

[19] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of TACAS'08, an ETAPS conference*, Apr. 2008, pp. 337–340.

[20] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. the 39th Int. Conf. on Dependable Systems and Networks (DSN'09)*, Jun. 2009.

[21] S. Muchnick and N. Jones, *Program Flow Analysis: Theory and Applications – Chapter 9 : L. Clarke, D. Richardson*. Prentice-Hall, 1981.

[22] S. Bardin and P. Herrmann, "Structural testing of executables," in *1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08)*, 2008, pp. 22–31.

[23] B. Elkarablieh, P. Godefroid, and M. Levin, "Precise pointer reasoning for dynamic test generation," in *Proc. of ISSTA*, 2009.

[24] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze, "Using symbolic execution for verifying safety-critical systems," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*. Vienna, Austria: ACM, September 2001, pp. 142–150.

[25] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. of 10th Workshop on Model Checking of Software (SPIN)*, 2003, pp. 235–239.

[26] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. on Soft. Eng. (TSE)*, vol. 30, no. 6, pp. 388–402, June 2004.

[27] E. Clarke and D. Kroening, "Hardware verification using ANSI-C programs as a reference," in *Proc. of ASP-DAC'03*, Jan. 2003, pp. 308–311.

En résumé, l'article de ce chapitre contient deux contributions au domaine du test à base de contraintes:

- un modèle mémoire pour le Bytecode Java permettant de modéliser l'héritage et le polymorphisme par invocation de méthodes virtuelles. Ce dernier point est traité avec l'introduction de contraintes sur les types pour la modélisation des opérations lié au raffinement des types. A notre connaissance, ce point n'avait jamais été traité de cette manière dans un modèle pour la génération automatique de données de test ;

- il propose l'exploration en arrière des programmes Java, qui en fait un outil idéal pour compléter une couverture de test obtenue par d'autres moyens de génération de données de tests. Les approches existantes de génération automatique de données de test qui s'appuient sur l'évaluation symbolique [Visser 04, Williams 05, Godefroid 05, Anand 07] proposent un parcours dans le sens avant des programmes, et peuvent donc se retrouver "piégées" en certain points du programme. A l'inverse la méthode que nous avons développée bénéficie de la réversibilité des contraintes et permet ainsi de parcourir l'arbre d'exécution du programme dans le sens inverse de l'exécution.

Chapter 7

Modélisation à contraintes des calculs flottants

Contexte

Depuis plusieurs années, la génération automatique de données d'entrée est devenue une composante essentielle de la boîte à outils du développeur d'applications. La construction de tests unitaires a été facilitée avec l'aide d'outils générant automatiquement des données d'entrée qui maximisent la couverture de code. Ces outils explorent symboliquement les chemins du programme sous test, en utilisant l'*évaluation symbolique dynamique* qui est une technique standard utilisée en test logiciel. Cette technique sélectionne un chemin en choisissant une donnée de test et en observant les instructions réellement exécutées. Le chemin ainsi caractérisé est nécessairement exécutable. Puis elle calcule la *condition de chemin* en évaluant symboliquement les instructions le long du chemin exécuté. En réfutant une décision sur la condition de chemin et en soumettant le système de contraintes ainsi obtenu à un résolveur de contraintes, cette technique peut soit produire une autre donnée de test qui couvre un chemin distinct, soit démontrer que le chemin correspondant au système de contraintes est non-exécutable. Lorsqu'une donnée de test est produite, la couverture des chemins du programme est nécessairement accrue.

Malheureusement, lorsque la condition de chemin contient des calculs sur les nombres flottants, la génération automatique de données d'entrée par évaluation symbolique dynamique devient problématique car la résolution de contraintes sur les réels ou les rationnels ne permet pas de donner des résultats corrects vis à vis des flottants. En effet, les erreurs d'arrondis perturbent les résolveurs de contraintes. Il est bien connu que l'évaluation d'une expression arithmétique sur les flottants peut donner des résultats très différents de son évaluation sur les réels ou les rationnels. Par exemple, la contrainte $(x > 0.0) \wedge (x + 1.0e12 == 1.0e12)$ où x est une variable flottante 32 bits, est satisfaite par tous les flottants 32 bits de l'intervalle $1.4012984643248171e-45 \ldots 32768.0$ tandis qu'il n'y a évidemment

aucune solution sur les nombres rationnels, ce que démontrent sans problème les résolveurs de contraintes existants. On pourrait argumenter que les erreurs d'arrondis des calculs flottants ne sont pas si fréquents, que les variables flottantes sont rarement utilisées pour piloter le flot de contrôle, et donc qu'elles peuvent être ignorées la plupart du temps. Cependant, les erreurs d'arrondi sont une source majeure de fautes graves dans les systèmes critiques, en particulier ceux du domaine militaire (e.g., l'accumulation des erreurs d'arrondi des missiles Patriot [Miné 04] ou bien les applications de suivi de terrain basse altitude des drones et avions militaires). Les techniques de test à base de contraintes n'échappent pas à ces problèmes.

Les travaux que nous avons initiés en 2000 dans le cadre du projet RNTL INKA ont conduit à la proposition de plusieurs approches qui visent à résoudre explicitement des contraintes sur les flottants. En 2001, Claude Michel, Michel Rueher et Yahia Lebbah ont proposé de bâtir un tel résolveur en utilisant la propagation d'intervalles [Davis 87]. L'idée de base consistait à trouver des bornes cohérentes pour les contraintes sur les flottants en pratiquant une exploration dichotomique de chaque intervalle [Michel 01]. En 2002, Claude Michel a proposé dans [Michel 02] des bornes théoriques pour atteindre une FP_2B_consistance, c'est à dire un niveau de filtrage pour les contraintes flottantes qui garantit que chaque borne d'intervalle a un support pour les autres variables. En 2003, le projet ACI V3F[1] (2003-2006) a été lancé afin d'étudier en profondeur la résolution de contraintes sur les flottants et ses différentes applications en génération automatique de données de test. De ce projet naquit non seulement le logiciel FPSE [Botella 05] présenté in extenso dans l'article [Botella 06] de ce chapitre, mais aussi le résolveur de contraintes sur les flottants de l'outil GATEL [Marre 05] développé par Bruno Marre au CEA.

Le but des travaux présenté dans l'article de ce chapitre est de modéliser fidèlement les erreurs d'arrondi dans un résolveur de contraintes sur les flottants. Le résultat est donc un résolveur qui prend en entrée une condition de chemin sur les flottants et évalue la satisfiabilité du système de contraintes sous-jacent. Si celui-ci est satisfiable, alors le résolveur propose une solution, qui peut être utilisée pour définir une donnée de test. Si celui-ci est insatisfiable, alors le chemin correspondant à la condition est démontré comme étant non-exécutable. Comme l'ensemble des valeurs flottantes pour une variable 32 ou 64 bits est fini (le standard actuel IEEE-754 propose seulement 4 types de flottants bornés [IEEE-754 85]), il est toujours possible dans le pire cas, d'énumérer toutes les combinaisons possibles de valeurs. Cependant, le véritable défi consiste à trouver des moyens efficaces pour couper l'espace de recherche car cette énumération est souvent hautement combinatoire. Bâtir une procédure de décision correcte et efficace sur les flottants est difficile puisque, d'une part, les propriétés algébriques des nombres flottants sont extrêmement pauvres (i.e., pas d'associativité, pas de distributivité) et, d'autre part, les résultats des calculs dépendent du compilateur et de ses options, ainsi que du matériel sous-jacent (processeur, jeu d'instructions, registres, etc.). La technologie utilisée dans le résolveur de contraintes sur les flottants FPSE est la propagation

[1] `http://lifc.univ-fcomte.fr/v3f/description.php`

d'intervalles et implémente une FP_2B_consistance. Les techniques d'énumération sont par contre similaires à celles que l'on trouve dans les résolveurs de contraintes sur les domaines finis.

B. Botella, A. Gotlieb, and C. Michel. *Symbolic execution of floating-point computations*. **The Software Testing, Verification and Reliability journal, 16(2):pp 97-121, June 2006**

Symbolic execution of floating-point computations ⋆

Bernard Botella [a], Arnaud Gotlieb [b,*], Claude Michel [c]

[a] *THALES Airborne Systems 2 av. Gay-Lussac 78851 Elancourt Cedex, FRANCE*

[b] *IRISA / INRIA Campus Beaulieu 35042 Rennes cedex, FRANCE*

[c] *I3S-CNRS 930, route des Colles, BP 154, 06903 Sophia Antipolis cedex, FRANCE*

Abstract

Symbolic execution is a classical program testing technique which evaluates a selected control flow path with symbolic input data. A constraint solver can be used to enforce the satisfiability of the extracted path conditions as well as to derive test data. Whenever path conditions contain floating-point computations, a common strategy consists of using a constraint solver over the rationals or the reals. Unfortunately, even in a fully IEEE-754 compliant environment, this leads not only to approximations but also can compromise correctness: a path can be labelled as infeasible although there exists floating-point input data that satisfy it. In this paper, we address the peculiarities of the symbolic execution of program with floating-point numbers. Issues in the symbolic execution of this kind of programs are carefully examined and a constraint solver is described that supports constraints over floating-point numbers. Preliminary experimental results demonstrate the value of our approach.

Key words: Symbolic execution, Floating-point computations, Automatic test data generation, Constraint solving

1 Introduction

Structural testing is usually required to find a test set that activates control flow paths that cover a selected testing criterion (e.g. all_statements, all_branches, ...). Introduced by King in the context of Software Testing [1], symbolic execution consists in statically evaluating statements of a program to find a test datum that activates a given control flow path. Input variables are replaced by symbolic input data and each statement of the path is evaluated by replacing internal references with an expression over the symbolic input data. Symbolic execution computes so-called path conditions that are constraints on the symbolic input data that characterize the selected path. Solving the path conditions permits input data to be obtained that activate the path. As only input values are generated, such an approach relies on the availability of an oracle. An oracle is just a procedure that checks the computed outcomes and produces a testing verdict. Symbolic execution can be used to address the path feasibility problem [2,3]. When the constraint set equivalent to the path conditions is unsatisfiable, then the selected path is shown to be infeasible. Note, however, that finding all the infeasible paths of a program is a classical undecidable problem [4]. Symbolic execution has been used in numerous applications, such as automatic structural test data generation [5,6,7,8,9,10,11], mutation-based testing [12], program specialization

⋆ This work is partially supported by the FNS granted project V3F

* Corresponding author.
Email address: `Arnaud.Gotlieb@irisa.fr` (Arnaud Gotlieb).

[13], parallelizing compilers [14], program and property proving [15,16], just to name a few.

Issues in floating-point computations. It is well known that reasoning over the reals or the rationals leads to some inconsistencies when the results are directly mapped over to the floating-point numbers [17]. In such a case, even in an environment which complies to the IEEE-754 standard for binary floating-point arithmetic [18], the symbolic execution of a program path which involves floating-point variables can produce not only inexact results but also incorrect ones. For example, consider the C program given in Fig.1 and the symbolic execution of path 1→2→3→4. The associated path conditions can be written as $\{x > 0.0, x + 1.0e12 = 1.0e12\}$. It is trivial to verify that these constraints do not have any solution over the reals or the rationals and a solver over the reals like the IC library of the Eclipse Prolog system [19] will immediately detect this. However, any IEEE-754 single-format floating-point numbers of the closed interval $[1.401298464324817e-45, 32767.9990234]$ is a solution of these path conditions. Hence, a symbolic execution tool working over the reals or the rationals will declare this path as being infeasible although this is clearly incorrect. Conversely, consider the path conditions $\{x < 10000.0, x + 1.0e12 > 1.0e12\}$ which could easily be extracted by the symbolic execution of path 1→2→3→4 of program foo2 of Fig.2. All the reals of the open interval (0, 10000) are solutions of these path conditions. However, there is no single floating-point value able to activate the path 1→2→3→4. Indeed, for any single floating-point number x_f in (0, 10000), we have $x_f + 1.0e12 = 1.0e12$ [1]. Hence the path 1→2→3→4 is actually infeasible although a symbolic execution tool over the reals or the rationals would

[1] This behaviour is called the addition *absorption*.


```
float foo1(float x) {
    float y = 1.0e12, z ;
1.  if (x > 0.0)
2.      z = x + y ;
3.  if (z == y)
4.      ...
```

Figure 1. Program foo1

```
float foo2(float x) {
    float y = 1.0e12, z ;
1.  if (x < 10000.0)
2.      z = x + y ;
3.  if (z > y)
4.      ...
```

Figure 2. Program foo2

have declared it as feasible.

This kind of behaviour can be obtained with any of the available solvers over the reals or the rationals. These solvers use a Linear Programming algorithm as in the clpr or in the clpq framework, or interval propagation with floating-point numbers to bound the reals such as in Ilog Solver, Eclipse IC [19], RealPaver [20] or Interlog [21,22]. The key issue here is that these solvers obey to mathematical rules which do not hold for floating-point arithmetic. As a matter of fact, floating-point arithmetic is quite poor. For example, with floating-point numbers, $x + (y + z)$ is not in general equal to $(x + y) + z$. Moreover, interval propagation based solvers assume that if $z = x + y$ then $x = z - y$. Unfortunately, due to rounding operations, this does not hold for floating-point arithmetic.

Such problems might be seen as unavoidable. By contrast, this paper introduces the techniques required to correctly handle these kinds of issues. Our approach is based on the following two steps:

- In a first step, complex expressions over the floating-point numbers are translated into equivalent relations which capture all the semantics of the floating-point operations; these relations are binary or ternary constraints

over the floating-point numbers.

- In a second step, a solver dedicated to floating-point numbers is used to solve the resulting constraints; this solver handles these constraints according to the semantics of floating-point arithmetic.

For example, consider again the path conditions extracted from Fig.1 and assume that the initial domain of variable x is $[-INF, +INF]$. The first constraint $x > 0.0$ reduces the interval of x to $[1.401298464324817e-45, +INF]$, the lower bound of which is the smallest non-zero positive number that can be represented in IEEE single-format floating-point arithmetic. Then, the second constraint $x+1.0e12 = 1.0e12$ reduces [2] the domain of x to $[1.401298464324817e-45, 32767.9990234]$. In this example, all the values of the resulting interval are solutions of the path conditions. Hence, it suffices to take any of the single floating-point of this interval to find a test datum that activates path 1→2→3→4 of the foo1 program. However, this is not generally the case and one must resort to enumeration to find a solution.

Contributions of the paper. This paper introduces new techniques to symbolically execute programs which involve floating-point computations. The paper extends the theoretical work of Michel [23] on the design of exact projection functions of constraints over the floating-point numbers. Practical details on how to build correct and efficient projection functions over floating-point intervals are given. The paper covers not only arithmetic operators but also comparison and format-conversion operators. FPSE, a symbolic execution tool for ANSI C floating-point computations, has been developed to validate the proposed approach. This paper describes its design and implementation and

[2] In IEEE-754 single-format, the constant $1.0e12$ is interpreted as 999999995904.

reports some initial experimental results. Note, however, that the paper does not address the general problem of testing floating-point computations. In particular, it does not study the difficult problem of obtaining a correct (but not necessarily exact) oracle in the presence of floating-point computations.

Contents. Section 2 briefly recalls the main principles of symbolic execution and reviews how several symbolic execution tools handle the problem of floating-point computations. Section 3 explains the essence of the IEEE-754 standard for binary floating-point arithmetic and indicates the limitations of the proposed approach. Section 4 presents the design of efficient projection functions over floating-point variables. Section 5 explains how to deal with symbolic values such as infinities. Section 6 describes FPSE and reports some experimental results. Finally, the last section describes directions for further work.

2 Related work

Only a few studies deal with floating-point computations in the Software Testing community. According to our knowledge, the only directly related work is that of Miller and Spooner [24]. Thirty years ago, they studied how to generate automatically floating-point test data for imperative programs. Their work opened the door for execution-based test data generation methods which does not suffer of the above mentioned problems. However, their approach makes only use of program executions and do not rely on symbolic reasoning. Thus, it cannot be used to study path feasibility.

At a time when no standard for floating-point arithmetic was available, sym-

bolic execution was pioneered by King [1], Clarke [5], Howden [6] and others [7,8,9] in several systems. SELECT [7] and DAVE [5] exploited Linear Programming algorithms to solve linear path conditions over the reals. CASEGEN [8] utilized ad-hoc procedures based on try-by-value methods to solve non-linear equations and used inequalities over the reals and to find test data that activated a selected path in the control flow graph. Although these systems were using floating-point operations in their computations, they solved path conditions over the reals. Thus, they did not conform to the floating-point computations of the program under test.

SMOTL [9] and more recently GODZILLA [12] took advantage of domain reduction techniques to prune the search space of integers inequalities. Gotlieb et al. [25] applied Constraint Logic Programming over finite domains to solve constraints extracted from imperative programs in the tool INKA [26]. The proposed framework dealt only with constraints over integers (possibly non-linear) to automatically generate test data. SMOTL, GODZILLA and INKA did not address the problem of floating-point computations in symbolic execution but they did use domain and interval propagation techniques to solve constraint systems. The method used in the current paper to solve path conditions over floating-point variables is closely related to these techniques.

More recently, Meudec followed a similar path in [11] and proposed solving path conditions over floating-point variables by means of a constraint solver over the rationals in the ATGen symbolic execution tool. The `clpq` library [27] of the Constraint Logic Programming system ECLIPSE was used to solve linear constraints over rationals computed with an arbitrary precision using an extended version of the simplex algorithm. Although this approach appears to be of particular interest in practice, it fails to handle correctly floating-point

computations.

Hence, the problem of floating-point computations in symbolic execution have not been seriously addressed in the past. Although several works deal with floating-point computations, none of them provide a correct handling of floating-point computations. Indeed, floating-point computation can be correctly handled neither with constraint solvers over the reals nor with constraint solvers over the rationals. Dealing with floating-point computations requires the development of a new constraint solver dedicated to floating-point numbers.

3 Preliminaries

This section introduces the arithmetical model specified by the IEEE-754 standard for binary floating-point arithmetic [18] and explains the limitations and notations of the proposed approach.

3.1 IEEE-754

IEEE-754 specifies two basic binary floating-point formats (single and double) and two extended formats. Each floating-point number is a triple (s, e, f) of bit patterns where s is the sign bit, e the biased exponent, and f the significand. The single format occupies 32 bits (1 bit for the sign, 8 for the exponent and 23 for the significant) while the double occupies 64 bits (1 bit for the sign, 11 for the exponent and 52 for the significant). The standard does not give a strict specification of the extended formats, but it does prescribe some minimal requirements over their sizes. For example, a double extended must occupy at least 79 bits. Each format defines several classes of numbers: nor-

malized numbers, denormalized numbers, signed zeros, infinities and NaNs (which stands for Not-a-Number). For the single format, normalized numbers corresponds to an exponent value $0 < e < 255$ and a value given by the formula: $(-1)^s\ 1.f\ 2^{e-127}$. Denormalized numbers correspond to an exponent $e = 0$ and a value given by $(-1)^s\ 0.f\ 2^{-126}$ where $f \neq 0$. Note that the significand possesses a hidden bit which is 1 for normalized numbers and 0 for denormalized. Note also that the bias is equal to 127 for the single format [3] and the exponent is -126 for denormalized numbers. There are two infinities (noted $+INF$, $-INF$ with $e = 255, f = 0$) and two signed zeros (noted $+0.0$, -0.0 with $e = 0, f = 0$) that allow certain algebraic properties to be maintained [17]. NaNs ($e = 255, f \neq 0$) are used to represent the results of invalid computations such as a division or a subtraction of two infinities. They allow the program execution to continue without being halted by an exception. IEEE-754 indicates four types of rounding directions: toward the nearest representable value, with "even" values preferred whenever there are two nearest representable values (to-the-nearest), toward negative infinity (down), toward positive infinity (up) and toward zero (chop). The most important requirement of IEEE-754 arithmetic is the accuracy of floating-point computations: each of the following operations, add, subtract, multiply, divide, square root, remainder, conversions and comparisons, must deliver to its destination the exact result if possible or the floating-point number that requires the least modification of the exact result w.r.t. the prescribed rounding mode and the result

[3] The actual value of the exponent is $E - bias$, where E is the exponent value in the floating-point number representation. Thus, with single format floating-point numbers, the maximum value of the exponent is 127 and the minimum value is -126.

format destination. It is said that these operations are correctly rounded [4]. For example, the single-format result of $999999995904 + 10000$ is [5] 999999995904 which is the single-format floating-point number nearest to the exact result over the reals. This example shows that the accuracy requirement of IEEE-754 does not prevent surprising results from arising (the second operand is absorbed by the addition operator).

3.2 Limitations and notations

In the sequel, we assume an IEEE-754 compliant floating-point unit. The types of floating-point numbers manipulated by the program are limited to the single and the double-format. The proposed framework currently handles only the to-the-nearest rounding direction, which is the default rounding mode in most programming languages. A decimal constant (such as $1.0e12$) denotes a floating-point value, and thus, has to be understood as the nearest floating-point number according to the default rounding mode (*i.e.* as 999999995904 with a to-the-nearest rounding mode). Zeros and infinities are handled but NaNs are not. Thus any floating-point unknown is assumed to take only a numerical or infinity value. Henceforth x^+ (resp. x^-) denotes the smallest (resp. greatest) floating-point number greater (resp. smaller) than x, with respect to its format. Moreover, $mid(a, b)$ denotes the floating-point number at the middle [6] of a and b. Finally, let $\oplus, \ominus, \otimes, \oslash$ denote floating-point operations

[4] IEEE-754 says equivalently "exactly rounded".

[5] These two decimals can be exactly represented by single binary floating-point numbers.

[6] which is a floating-point number of a wider format than the one of its two operands.

(*i.e.* the format dependent result of a to-the-nearest rounding of the exact result) whereas $+, -, *, /$ denote the same operations over the reals. This paper addresses only the problem of dealing with floating-point variables in symbolic execution; other issues such as dealing with loops, arrays and pointers in symbolic execution are out of the scope of this paper. These problems are more detailed in [28,29,10,11,30]. Finally, the combination of integers and floating-point expressions into a symbolic execution framework are not detailed here. Hence, programs are limited to floating-point data types.

4 Symbolic execution

Symbolic execution has been formally described by Clarke and Richardson in [28]. This technique is based on the selection of a single path of the control flow graph and the computation of symbolic states. When one has to deal with floating-point computations, special attention must be paid to the way expressions are evaluated, as described in this section.

4.1 Control flow graph and paths

The control flow graph of a program P is a connected oriented graph composed of a set of vertices, a set of edges and two distinguished nodes, e the unique entry node, and s the unique exit node. Each node represents a basic block and each edge represents a possible branching between two basic blocks. A path of P is a finite sequence of edge-connected nodes of the control flow graph which starts on e. $Var(P)$ denotes the set of variables of P.

4.2 Symbolic states and expressions

Symbolic execution works by computing symbolic states for a given path. A *symbolic state* for path $e \rightarrow n_1 \rightarrow \ldots \rightarrow n_k$ in P is a triple $(e \rightarrow n_1 \rightarrow \ldots \rightarrow n_k, \{(v, \phi_v)\}_{v \in Var(P)}, c_1 \wedge \ldots \wedge c_n)$ where ϕ_v is a symbolic expression associated to the variable v and $c_1 \wedge \ldots \wedge c_n$ is a conjunction of symbolic expressions, called path conditions. A *symbolic expression* is either a symbolic value (possibly **undef**) or a well parenthesized expression composed over symbolic values. In fact, when computing a new symbolic expression, each internal variable reference is replaced by its previous symbolic expression. For example, the symbolic state of path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ in the program of Fig. 1 can be obtained by the following sequence of symbolic states :

$(1 \rightarrow 2,\ \{(x, X), (y, 1.0e12), (z, \mathbf{undef})\}, X > 0.0)$

$(1 \rightarrow 2 \rightarrow 3,\ \{(x, X), (y, 1.0e12), (z, X \oplus 1.0e12)\}, X > 0.0)$

$(1 \rightarrow 2 \rightarrow 3 \rightarrow 4, \{(x, X), (y, 1.0e12), (z, X \oplus 1.0e12)\}, X > 0.0 \wedge X \oplus 1.0e12 == 1.0e12)$

where X is the symbolic value of the input variable x.

Usually, symbolic expressions and path conditions hold only over symbolic input values. However, when floating-point computations are involved in the path, other symbolic values can appear in the symbolic expressions, as described below.

4.3 Forward/backward analysis

Symbolic states are computed by induction on their path by a forward or a backward analysis [28]. Each statement of each node of the path is symbolically evaluated using an evaluation function which computes the symbolic

states. Forward analysis follows the statements of the selected path in the same direction as that of actual program execution, whereas backward analysis uses the reverse direction. Backward analysis is usually preferred when one only wants to compute the path conditions.

4.4 Normalization

In the presence of floating-point computations, special attention must be paid to conform to the actual execution of program. It is necessary to take into account the evaluation order and the precedence of expression operators as specified by the language [7]. The idea is to exploit the expression's shape of the abstract syntax tree built by the compiler of the program without any rearrangement nor any simplification due to optimizations [8]. When symbolic expressions are directly extracted from the abstract syntax tree then, not only the operator precedence is respected but also is the order in which operands are evaluated. This is not always the case when symbolic expressions are extracted from source code by an analyzer. Preserving the order of evaluation in the analyzer is essential with floating-point computations as simple algebraic properties such as associativity or distributivity are lost. An approach called normalization is proposed here. It decomposes expressions and takes into account the above requirements. Normalization makes symbolic expressions over the floating-point numbers independent from the compiling environment.

[7] Some languages are quite permissive and give to the compiler some freedom in the interpretation of floating-point expressions. In such a case, we have to observe the actual behaviour of the compiler.

[8] Compiler optimisation flags are not allowed here particularly when they rearrange instructions.

Any of the symbolic expressions is decomposed in a sequence of assignments where fresh temporary variables [9] are introduced bearing in mind that the order of evaluation must be preserved. For example, let $E = v_1 \otimes v_2 \otimes v_3 \oplus v_4$ then the resulting decomposition is $E = t_1 \oplus v_4 \wedge t_1 = t_2 \otimes v_3 \wedge t_2 = v_1 \otimes v_2$ because $\otimes$ has a higher priority than $\oplus$ and operands are evaluated from left to right. This decomposition requires that intermediate results of an operation conform to the type of storage of its operands [10]. In the previous example, if v_1 and v_2 are of single-format, then the temporary variable t_2 must also be single-format. As a result, path conditions are only composed of binary or ternary symbolic expressions that have a single operator over a known floating-point format. This form is called the normalized form of a symbolic expression.

5 Solving path conditions over the floating-point numbers

In this section, the floating-point variables are supposed to take a numerical value. We assume here that the computations do not overflow or raise exceptions. These behaviours are handled by means of infinites and NaNs and will be considered in the next section.

Path conditions are composed of normalized symbolic expressions over floating-

[9] The introduction of temporary variables does not change the semantic of floating-point computations as long as it maps the behaviour of the compiler and of the floating-point unit.

[10] This property is not a requirement of IEEE-754 and consequently it is not always true. For example, on Intel's architectures extended formats are used by default to store intermediate results

point input and temporaries variables. Each of these variables takes its numerical values within a finite interval of possible floating-point values w.r.t. its format. Intervals are represented by a couple of bounds that can possibly be provided by the user. By default, any numerical single-format floating-point values belongs to $[-3.40282347e38,\ 3.40282347e38]$ and any double-format values belongs to $[-1.7976931348623158e308,\ 1.7976931348623158e308]$.

5.1 Interval propagation

The solving process is based on interval propagation [31,32], which is a classical technique used to compute the set of solutions of non-linear constraints over the reals. The technique takes advantage of interval arithmetic [33] and relational arithmetic [34] to reduce the domains of the variables. If $I_x = [a, b]$ and $I_y = [c, d]$ then interval arithmetic says that $I_{x+y} = [a+c, b+d]$ contains all possible values for the expression $x + y$ when $x \in I_x$ and $y \in I_y$. In the same way, $I_{x-y} = [a-d, b-c]$, $I_{x*y} = [min(a*c, a*d, b*c, b*d), max(a*c, a*d, b*c, b*d)]$, $I_{exp(x)} = [exp(a), exp(b)]$, etc. Relational arithmetic allows decomposing the constraints in projection functions over intervals. For example, the constraint $z = x + y$ is decomposed into three projection functions:

$$I_z \leftarrow I_{x+y} \cap I_z, \quad I_x \leftarrow I_{z-y} \cap I_x, \quad I_y \leftarrow I_{z-x} \cap I_y$$

A constraint propagation algorithm uses these projection functions to compute a conservative approximation of the solutions of the constraint system. The following example of a constraint system over the reals illustrates this technique.

Example 1 *Let $x \in (-\infty, +\infty), y \in (-\infty, +\infty)$ be two real unknowns in the constraint system $y = log(x), x + y = 0$. After a decomposition of the constraints into*

projection functions, the following successive approximations of x and y are obtained by interval propagation :

$$x \in (-\infty, +\infty) \quad x \in [0, +\infty) \quad x \in [0, 1] \quad x \in [0.56, 1] \quad x \in [0.56, 0.57] \quad \ldots$$

$$y \in (-\infty, +\infty) \quad y \in (-\infty, 0] \quad y \in [-1, 0] \quad y \in [-1, -0.56] \quad y \in [-0.57, -0.56] \quad \ldots$$

Interval propagation has been applied in several systems [35,32] and two authors of the present paper contributed to the development of one of them, namely Interlog [21,22]. The work presented here mainly consists in adapting a real-based interval propagation system to floating-point numbers. It essentially requires modifying projection functions to handle conservatively the domains of floating-point variables. In the next subsections, interval propagation of floating-point intervals and projection functions for floating-point constraints are described.

5.2 Propagation over floating-point intervals

During interval propagation, projection functions are incrementally introduced into a *propagation queue*. An iterative algorithm manages each function one by one into this queue by filtering the domains of floating-point variables of their inconsistent values. Filtering algorithms consider only the bounds of the domains to eliminate inconsistent values. When the domain of a variable has been narrowed then the algorithm reintroduces in the queue all the projection functions in which this variable appears in order to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fixpoint).

When selected in the propagation queue, each function is added into a *constraint–store*. The constraint–store is contradictory when the domain of at least one

variable becomes empty during the propagation. In this case, the set of constraints (path conditions) is known to be unsatisfiable and the corresponding path is shown to be infeasible. The interval propagation process reaches a fixpoint because only a finite number of floating-point values can be removed from the domains. This fixpoint is a conservative overestimation (Cartesian product of intervals) of the possible floating-point values for the input variables.

As is usually the case with interval propagation solvers, propagation over floating-point intervals does not ensure that the set of constraints is satisfiable when a fixpoint is reached. Hence, one must resort to enumeration to locate particular solutions. This is done by a *labelling procedure* which tries to systematically assign a floating-point to a variable and initiate propagation through the constraint–store. This process is repeated until all the uninstantiated variables become bound. If this valuation leads to a contradiction then the process backtracks to other possible values or variables.

5.3 Floating-point variable projections

In the proposed approach, each normalized symbolic expression is decomposed into ternary and binary symbolic expressions. These expressions could be directly translated into elementary constraints. Each of these constraints is a ternary or binary constraint and is itself decomposed into projection functions. A ternary symbolic expression $r = a \odot b$ where $\odot$ denotes one of the four arithmetical operations $\oplus, \ominus, \otimes, \oslash$, is decomposed into 3 projections: the direct projection $proj(r, r = a \odot b)$, the first inverse projection $proj(a, r = a \odot b)$ and the second inverse projection $proj(b, r = a \odot b)$. Inverse means that projection is performed on a right operand of an assignment. The variable a in $proj(a, r = a \odot b)$ is called the projected variable. Note that single assignment $r = a$ can be treated as the ternary symbolic expression $r = a \ominus +0.0$ because $a \ominus +0.0 = a$ even when $a = -0.0$. A binary symbolic expression $a = (type)b$ where $type$ is either $float$ or $double$ is decomposed into a direct projection $proj(a, a = (type)b)$ and an inverse one $proj(b, a = (type)b)$. A binary symbolic expression $a\ rel\ b$ where rel denotes any of the six relational operators $==, <, =<, >, >=, !=$ is decomposed into two projections : $proj(a, a\ rel\ b)$ and $proj(b, a\ rel\ b)$.

5.3.1 Computing direct projections for ternary symbolic expressions

Let $[r_l, r_h], [a_l, a_h], [b_l, b_h]$ be the current floating-point domains of r, a, b, then the direct projection $proj(r, r = a \odot b)$ computes new bounds r'_l, r'_h for the domain of r by using the formula of Fig.3.

$$
\begin{array}{ll}
[r'_l, r'_h] \leftarrow [a_l \oplus b_l, a_h \oplus b_h] \cap [r_l, r_h] & \text{when } \odot = \oplus \\
[r'_l, r'_h] \leftarrow [a_l \ominus b_h, a_h \ominus b_l] \cap [r_l, r_h] & \text{when } \odot = \ominus \\
[r'_l, r'_h] \leftarrow [min(a_l \otimes b_l, a_l \otimes b_h, a_h \otimes b_l, a_h \otimes b_h), max(a_l \otimes b_l, a_l \otimes b_h, a_h \otimes b_l, a_h \otimes b_h)] \cap [r_l, r_h] & \\
 & \text{when } \odot = \otimes \\
[r'_l, r'_h] \leftarrow [min(a_l \oslash b_h, a_l \oslash b_l, a_h \oslash b_h, a_h \oslash b_l), max(a_l \oslash b_h, a_l \oslash b_l, a_h \oslash b_h, a_h \oslash b_l)] \cap [r_l, r_h] & \\
 & \text{when } \odot = \oslash \text{ and } +0.0, -0.0 \text{ do not belong to } [b_l, b_h]
\end{array}
$$

Figure 3. Formulae for direct projections $proj(r, r = a \odot b)$

Although this remains implicit, it is important to bear in mind that these formulae are based on the to-the-nearest rounding mode. Note also that they was inspired by interval arithmetic [33,36] but differ from it [11]. Thanks the

[11] For example, the expected result over the reals of the sum of two numbers x and y can be captured by the interval $[\underline{z}, \overline{z}]$ where $\underline{z}$ (resp. $\overline{z}$) denotes the rounded toward negative (resp. positive) infinity result of $x + y$ [17].

monotonicity of the to-the-nearest rounding direction, these formula can directly be deduced from the interval arithmetic. The special case where $+0.0$ or -0.0 belongs to the right operand of the $\oslash$ operator can easily be handled by using infinities; this will be explained in the next section. Note also that the intersection of two intervals can be computed by using the formula $[a,b] \cap [c,d] = [max(a,c), min(b,d)]$ as the set of **numerical** floating-point values is totally ordered (even for both -0.0 and $+0.0$). Fig. 4 shows an example of application of the formula for the operator $\oplus$. The intervals of a, b, r are shown with vertical lines and each horizontal arrow represents the actual computation of the new bounds of r, before rounding. In this example, the new inferior bound of r is rounded up although the result over the reals $a_l + b_l$ is strictly less than the to-the-nearest rounded result of $a_l \oplus b_l$. This is due to the fact that $a_l + b_l$ is strictly greater than $mid((a_l \oplus b_l)^-, a_l \oplus b_l)$. This shows that the formula does not usually retain the solutions over the reals but handles all the solutions over the floating-point numbers.

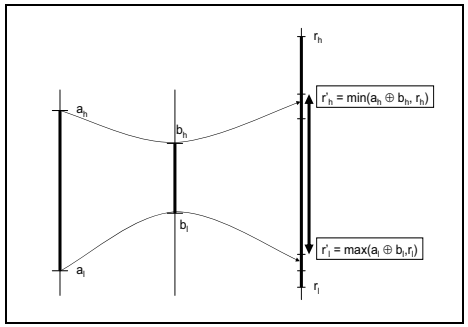


Figure 4. Computation of direct projection $proj(r, r = a \oplus b)$

Note that these formula for direct projections lead to an optimal pruning of the interval of r, because IEEE-754 guarantees that the four arithmetic operations are correctly rounded.

5.3.2 *Computing inverse projections*

Inverse projections are a little bit more complicated to compute. The **first inverse** projection $proj(a, r = a \odot b)$ computes new bounds a'_l, a'_h for the domain of a whereas the **second inverse** projection $proj(b, r = a \odot b)$ computes new bounds b'_l, b'_h for the domain of b. The formulae to compute these inverse projections are given in Fig. 5. Note that the first and the second projections for $\oplus$ and $\otimes$ are the same. Thus, only one of them is given here.

$$[a'_l, a'_h] \leftarrow [mid(r_l, r_l^-) \ominus b_h, mid(r_h, r_h^+) \ominus b_l] \cap [a_l, a_h] \qquad \text{when } \odot = \oplus$$

$$[a'_l, a'_h] \leftarrow [mid(r_l, r_l^-) \oplus b_l, mid(r_h, r_h^+) \oplus b_h] \cap [a_l, a_h] \qquad \text{when } \odot = \ominus \text{ (first inverse)}$$

$$[b'_l, b'_h] \leftarrow [a_l \ominus mid(r_h, r_h^+), a_h \ominus mid(r_l, r_l^-)] \cap [b_l, b_h] \qquad \text{when } \odot = \ominus \text{ (second inverse)}$$

$$[a'_l, a'_h] \leftarrow \begin{array}{l} [min(mid(r_l, r_l^-) \oslash b_l, mid(r_l, r_l^-) \oslash b_h, mid(r_h, r_h^+) \oslash b_l, mid(r_h, r_h^+) \oslash b_h), \\ max(mid(r_l, r_l^-) \oslash b_l, mid(r_l, r_l^-) \oslash b_h, mid(r_h, r_h^+) \oslash b_l, mid(r_h, r_h^+) \oslash b_h)] \end{array} \cap [a_l, a_h]$$

when $\odot = \otimes$ and $+0.0, -0.0$ do not belong to $[b_l, b_h]$

$$[a'_l, a'_h] \leftarrow \begin{array}{l} [min(mid(r_l, r_l^-) \otimes b_l, mid(r_l, r_l^-) \otimes b_h, mid(r_h, r_h^+) \otimes b_l, mid(r_h, r_h^+) \otimes b_h), \\ max(mid(r_l, r_l^-) \otimes b_l, mid(r_l, r_l^-) \otimes b_h, mid(r_h, r_h^+) \otimes b_l, mid(r_h, r_h^+) \otimes b_h)] \end{array} \cap [a_l, a_h]$$

when $\odot = \oslash$ (first inverse)

$$[b'_l, b'_h] \leftarrow \begin{array}{l} [min(a_l \oslash mid(r_l, r_l^-), a_h \oslash mid(r_l, r_l^-), a_l \oslash mid(r_h, r_h^+), a_h \oslash mid(r_h, r_h^+)), \\ max(a_l \oslash mid(r_l, r_l^-), a_h \oslash mid(r_l, r_l^-), a_l \oslash mid(r_h, r_h^+), a_h \oslash mid(r_h, r_h^+))] \end{array} \cap [b_l, b_h]$$

when $\odot = \oslash$ (second inverse) and $+0.0, -0.0$ do not belong to $[b_l, b_h]$

Figure 5. Formula for inverse proj. $proj(a, r = a \odot b)$ and $proj(b, r = a \odot b)$

First, all inverse projections computes the middle of (r_l, r_l^-) and the middle of (r_h, r_h^+). The reason for that is that r is the result of a to-the-nearest rounding. More precisely, as the implemented operations are correctly rounded, they might be seen as the rounding to to-the-nearest of the result $r_{\mathbb{R}}$ over the reals of the same operation over the reals. Thus, if the floating point number r_l is the result of a to-the-nearest rounding, $r_{\mathbb{R}}$ has to belong to the interval [12]

[12] $[mid(r_l, r_l^-), mid(r_l, r_l^+)]$ is a conservative overestimation. A more precise interval could be computed if we take into account the value of the least significant bit of r_l (or r_h).

$[mid(r_l, r_l^-), mid(r_l, r_l^+)]$. The same reasoning applies to r_h. The computation of the middle of two single-format or double format floating-point variables can easily be computed as a wider format is almost always available [13]: the middle of two singles is captured by a double and the middle of two doubles is captured by an extended double. Note that the operations themselves are performed over a wider format, such as in the inverse projection of $\oplus$: $mid(r_l, r_l^-) \ominus b_h$ as shown in Fig. 6. Here, both operands of $\ominus$ are first converted into a greater format, although this remains implicit in the formula.

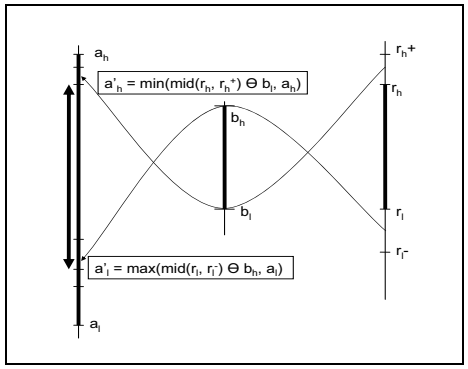


Figure 6. Computing first inverse projection $proj(a, r = a \oplus b)$

Second, special attention must be paid to the computation of the bounds of the projected variable. Operators $\oplus, \ominus, \otimes, \oslash$ are correctly rounded. Thus, they can be used to compute their inverse. The complete proof of this statement can be found in [23] and only an outline of it is given here. Consider the computation of a'_h for the addition in Fig. 6. As explained above, r_h is the

[13] Note however that an overestimation of the solution can still be computed using the same format as the operands, but this usually leads to a greater imprecision. For example, $[a'_l, a'_h] \leftarrow [r_l^- \ominus b_h, r_h^+ \ominus b_l] \cap [a_l, a_h]$ is a conservative overestimation for the first inverse projection of the addition.

result of a to-the-nearest rounding of the addition of a' and b over the reals. Thus, over the reals, the following inequality holds : $a'_h + b \leq mid(r_h, r_h^+)$ where the floating-point number $b \in [b_l, b_h]$. Over the reals, this inequality leads to $a'_h \leq mid(r_h, r_h^+) - b_l$. In order to obtain a'_h, that is to say, in order to find the greatest floating-point number less or equal to $mid(r_h, r_h^+) - b_l$ (which is nothing but the definition of a rounding to $-\infty$), we would have to compute $mid(r_h, r_h^+) - b_l$ with a rounding to $-\infty$. However, a to-the-nearest rounding computes a conservative value for a'_h, *i.e.* a value that is equal or greater than the optimal value, and avoid the cost of a modification of the rounding mode.

As a consequence, the formula given here for computing the inverse projections are not always optimal but offer a conservative overestimation of the set of floating-point values that satisfy a given normalized symbolic expression. Considering the least significant bit of r_l and r_h can lead to slightly more shrinking [23] but requires changing the rounding mode several times during the computation of each projection function. Note also that interesting results from the literature can be used to improve the computation of inverse projections. For example, a classical result [37] says that if $x \oplus y$ underflows to a denormalized number then $x \oplus y$ is exactly equal to $x + y$. In such a case, the computation of the middle $mid(rh, rh^+)$ might be avoided.

5.4 Handling comparisons and conversions

Comparisons. Relational operators such as $==, >, >=, <, <=, !=$ are handled by ordered set properties because the finite set of numerical floating-point variables is totally ordered. The formula is similar for the first and the second projections, hence only the first are given in Fig. 7. The floating-point domain

of a (resp. b) is $[a_l, a_h]$ (resp. $[b_l, b_h]$) and the domain of the result a' is $[a'_l, a'_h]$.

$$
\begin{array}{ll}
[a'_l, a'_h] \leftarrow [max(a_l, b_l), min(a_h, b_h)] & \text{for } proj(a, a == b) \\
[a'_l, a'_h] \leftarrow [max(a_l, b_l), a_h] & \text{for } proj(a, a \geq b) \\
[a'_l, a'_h] \leftarrow [max(a_l, b_l)^+, a_h] & \text{for } proj(a, a > b) \\
[a'_l, a'_h] \leftarrow [a_l, min(a_h, b_h)] & \text{for } proj(a, a \leq b) \\
[a'_l, a'_h] \leftarrow [a_l, min(a_h, b_h)^-] & \text{for } proj(a, a < b) \\
[a'_l, a'_h] \leftarrow [\text{if } (a_l = b_l = b_h) \text{ then } a_l^+ \text{ else } a_l, & \\
\qquad \text{if } (a_h = b_l = b_h) \text{ then } a_h^- \text{ else } a_h] & \text{for } proj(a, a!{=}b)
\end{array}
$$

Figure 7. Formulae for projections coming from comparison operators

These formulae are mainly inspired by interval arithmetic [33] but slightly differ from it for the computation of modified bounds. Here, the computation benefits from the fact that it operates over a finite set of floating-point values. **Conversions.** The simple language described in Sec.3.2 allows only two conversions $r = (float)a$ where a is a double and $r = (double)a$ where a is a single. Formulae that compute the bounds of projected variables with direct and inverse projections of conversion operators are given in Fig. 8. Note that any single-format value can be exactly converted into a double-format value. Thus, some conversions do not require any computation and remain implicit in the formulae.

$$
\begin{array}{lll}
[r'_l, r'_h] \leftarrow [max\_f((float)a_l, r_l), min\_f((float)a_h, r_h)] & \text{for} & proj(r^f, r^f = (float)a^d) \\
[a'_l, a'_h] \leftarrow [max\_d(a_l, mid(r_l, r_l^-)), min\_d(a_h, mid(r_h, r_h^+))] & \text{for} & proj(a^d, r^f = (float)a^d) \\
[r'_l, r'_h] \leftarrow [max\_d(a_l, r_l), min\_d(a_h, r_h)] & \text{for} & proj(r^d, r^d = (double)a^f) \\
[a'_l, a'_h] \leftarrow [max\_f(a_l, r_l), min\_f(a_h, r_h)] & \text{for} & proj(a^f, r^d = (double)a^f)
\end{array}
$$

where r^f, a^f denote single-format variables, and r^d, a^d denote double-format variables,

max_f, min_f operate over the singles and max_d, min_d operate over the doubles

Figure 8. Formulae for projections coming from conversion operators

6 Handling symbolic values

IEEE-754 distinguishes two kinds of symbolic values: infinities and NaNs. The cases where infinities and NaNs can be produced as the result of a computation are detailed in [17]. However, implementing projection functions over symbolic values requires to further analysis of how to combine infinities, numerical values, zeros and NaNs and how to deal with exceptions [37].

In the proposed approach, the numerical domain is merely extended with both infinities and remains totally ordered. Roughly speaking, the main idea for computing projections consists in isolating the infinities from the numerical values of the domains, computing the projected variable's domain in the numerical case, combining the symbolic values between themselves, and merging the results of both the symbolic and the numerical cases.

To compute the projections of $\oplus$ (direct and inverse), tables 1 and 2 are required. Note that Nv stands for any non-zero numerical value and $\pm INF$ denotes any of the two infinities.

Table 1

Value of r in direct $proj(r, r = a \oplus b)$

$a \backslash b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$-INF$	$-INF$	$-INF$	$-INF$	$\perp$
-0.0	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$+0.0$	$-INF$	$+0.0$	$+0.0$	Nv	$+INF$
Nv	$-INF$	Nv	Nv	$Nv \cup \{\pm INF, +0.0\}$	$+INF$
$+INF$	$\perp$	$+INF$	$+INF$	$+INF$	$+INF$

Some combinations of symbolic values are impossible. For example, when $r = +0.0$ and $b = +INF$, the first inverse projection $proj(a, r = a \oplus b)$ computes an empty domain for variable a. Thus, there exists no floating-point value of

a able to satisfy the equation $+0.0 = a \oplus +INF$. These cases are indicated by the presence of the symbol $\perp$. When the operands of a projection are known and $\perp$ is encountered in the tables then the projection is refuted and the constraint store is shown to be contradictory. Note that when the sum of two opposite operands is exactly zero and the rounding mode is the to-the-nearest mode, then the result is $+0.0$ (and not -0.0). The cases where infinity is produced as the result of an operation over two numerical values (such as in $Nv \oplus Nv$) usually correspond to an overflow.

More frequently, operands are just known by their interval of possible values. Hence, when a combination of bounds is $\perp$, such as in $proj(a, r = a \oplus b)$ where $r \in [-INF, +INF]$ and $b \in [-INF, +INF]$, $\perp$ is just ignored and the interval of a is leaved unchanged (although $+0.0$ belong to the interval of r). The new bounds of r are computed using the formula of the numerical case ($[r'_l, r'_h] \leftarrow [a_l \oplus b_l, a_h \oplus b_h] \cap [r_l, r_h]$). Signed zeros, infinities and overflows are just special cases of this computation. If signed zeros belongs to the intervals of a or b then the numerical case ($Nv \oplus Nv$) of the table is applied. If an overflow occurs then the bounds are updated with the corresponding infinities.

Table 2

Value of a in first inverse $proj(a, r = a \oplus b)$

$_b \backslash^r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$Nv \cup \{-INF, \pm 0.0\}$	$\perp$	$\perp$	$\perp$	$\perp$
-0.0	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$+0.0$	$-INF$	$\perp$	± 0.0	Nv	$+INF$
Nv	$Nv \cup \{-INF\}$	$\perp$	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{+INF\}$
$+INF$	$\perp$	$\perp$	$\perp$	$\perp$	$Nv \cup \{+INF, \pm 0.0\}$

The same procedure can be used for the computation of the projections of $\ominus, \otimes, \oslash$ using the tables given at the end of the paper. Note that the negative and positive numerical cases have not been distinguished in these tables. Although this is useful to implement better pruning of domains, these cases are not difficult to determine as simple sign rules remain valid in the context of non-zeros numerical floating-point values. Note that the only cases where NaN is produced when operands are non-NaNs are $\infty - \infty$ for $\oplus, \ominus$ and $0 * \infty, 0/0, \infty/\infty$ for $\otimes$ and $\oslash$.

7 A labelling procedure

As previously said, projection functions only reduce the domains of the variables. Thus, constraint propagation ensures neither the path conditions are satisfiable nor a test datum to be found in the general case. Note however that this process is efficient as it only requires $O(md)$ operations in the worst case where m denotes the number of constraints and d denotes the size of the largest domain [22]. To find a solution, a labelling procedure has to be implemented. Some heuristics are used to choose the variables and the values to be first enumerated. Several heuristics have been discussed in [38] and can easily be implemented. Note that in a symbolic execution framework, only the input variables need to be instantiated as all the other internal variables are computed in terms of these. As soon as a value is given to an uninstantiated variable, the interval propagator wakes up all the projection functions where this variable appears, thereby propagating the choice through the constraint system. In the applications of symbolic execution over floating-point variables, two difficult situations may sometimes occur at the end of the initial propagation step: either the path conditions have no solutions (*i.e.* the corresponding path is non-feasible) but this has not been detected, or the path conditions

have solutions but the resulting intervals are too approximate for it to be found. In these two related situations the labelling process is time-consuming and cannot be completed in all the cases. However, note there are always less than 2^{32} (resp. 2^{64}) possible values in the domain of a single-format (resp. double-format) floating-point value. So the process is no more time-consuming than the one used in constraint-based automatic test data generation environments over integers [39,25,11].

8 Implementation and experimental results

We implemented a symbolic execution tool for ANSI C floating-point computations, called FPSE (Floating Point Symbolic Execution). The tool extracts path conditions and symbolic expressions by a forward analysis and tries to solve them using the principles described in this paper. The constraint propagation engine of FPSE is written in Prolog whereas the projection functions are written in C.

FPSE handles floating-point computations that strictly conform to IEEE-754 and are intended to run on Sparc architectures. ANSI C accommodates the IEEE 754 floating point standard by not adopting any constraints on floating point which are contrary to this standard. In particular, it allows operations on `float` to be performed in single precision calculations. Note, however, that ANSI C gives the compiler a large degree of freedom in how to interpret and evaluate a floating-point expression to a precision wider than that normally associated with its type. While compiling the tested programs, it is necessary to avoid the use of compiler options that activate code optimizations as well as options that allow the storage of floating-point values into extended formats.

In practice, it is very difficult to guarantee that the symbolic execution will strictly conform to the actual execution because of several reasons: the lack of documentation of the compiler options and design, the existence of unexpected hardware optimizations such as the fused multiply-add a+b*c, the unexpected change of rounding modes by user actions, the defaults in the compiler implementation and so on. These limitations have to be taken into account when interpreting the results of FPSE.

8.1 Experimental results

To evaluate the approach, we compared the results provided by FPSE with expected floating-point results computed by hand and results obtained with three available solvers over the reals and the rationals. Distributed as part as the ECLIPSE Prolog system, are the following three distinct solvers:

(1) the *IC* library [19] which is an hybrid integer/real interval arithmetic constraint solver based on interval propagation. As in any other interval propagation solver **over the reals** (*e.g.* Ilog solver, RealPaver [20], Interlog [21,22]), each real number is represented by a pair of floating point bounds and any arithmetic operation is performed by using these bounds. The resulting interval is then widened to take into account any possible error in the operation, thus ensuring the resulting interval contains the true answer over the reals. This contrasts with our approach where floating-point numbers and operations are correctly approximated by relations over finite sets of floating-point numbers (also represented by pair of floating point bounds);

(2) the *clpr* library [27] that solves linear constraints **over the reals**. *clpr*

makes use of floating-point numbers to approximate computations over the reals;

(3) the *clpq* library [27] that solves linear constraints **over rationals** computed with an arbitrary precision. In *clpq*, each rational is treated as a pair of integers and any arithmetic computation remains exact;

Both solvers *clpr* and *clpq* exploit the simplex method and a Fourier-Motzkin algorithm to solve linear constraints. In addition, they provide several isolation axioms to take into account some restricted shapes of non-linear constraints.

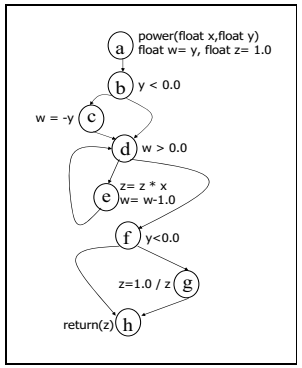


Figure 9. Control flow graph of program power.c

Programs. Several floating-point programs of small size were extracted from the literature to be carefully examined. We considered two distinct uses of symbolic execution: output symbolic expression computation and path feasibility.

Firstly, symbolic expressions were extracted from [17] and implemented in programs `g1.c, g2.c`. `g1.c` contains the C expression $X = ((2.0e-30 + 1.0e30) - 1.0e30) - 1.0e-30$ whereas `g2.c` contains $\delta == B^2 - 4AC$. For this

latter, two symbolic expressions were computed: the first one corresponds to **the direct evaluation** of the expression by taking $A = 1.22, B = 3.34, C = 2.28$ whereas the second one corresponds to **the inverse evaluation** where C is unknown and $\Delta == +0.0$. Symbolic expressions were extracted from paths of the program `power.c` that computes x^y, given in Fig.9. The two selected paths contain a number of iterations (40 and 350) that lead to overflows. All these symbolic expressions are given in the top of Tab.3.

Secondly, path feasibility was experimented with FPSE on path conditions extracted from programs `foo1.c` and `foo2.c` given in the introductory part of the paper (Fig.1,2), from the program `howden.c` that is a small-size numeric computation extracted from [40] and from the program `power.c` (Fig.9). For these programs, path conditions are given in the bottom of Tab.3. Second column provides the expressions as they appear in the literature. In particular, note that the path conditions of examples 8,9,10,11 results from a simplification process which has eliminated several redundant constraints. This process, as proposed for several symbolic execution tools [10], is unsound over floating-point variables as algebraic properties (such as associativity and distributivity) are not preserved. Third column of Tab.3 contains the normalized symbolic expressions as they are computed by the FPSE tool in the normalization process (sec.4). Finally, the last column contains the number of constraints present in the normalized path conditions.

All programs were compiled with `gcc`[14] on an ultra Sparc FPU under Solaris 2.7.

[14] `gcc-3.3.3 -g -Wall -DFPSE_SPARC -lm -std=gnu89 -ffloat-store -mhard-float -msoft-quad-float -munaligned-doubles` (some default options)

Table 3

Programs and FPSE expressions

	Program	Symbolic expr. over $\mathbb{R}$	Normalized FPSE expression	#
1	g1.c	$X = (2 * 10^{-30} + 10^{30}) - 10^{30} - 10^{-30}$	$T_2 = 2.0e-30 \oplus 1.0e30, T_1 = T_2 \ominus 1.0e30,$ $X = T_1 \ominus 1.0e-30$	3
2	g2.c	$\Delta = B^2 - 4 * A * C$ with $A = 1.22, B = 3.34, C = 2.28$	$T_1 = B \otimes B, T_2 = A \otimes C, T_3 = 4.0 \otimes T_2,$ $\Delta = T_1 \ominus T_3$	4
3	g2.c	$\Delta = B^2 - 4 * A * C$ with $A = 1.22, B = 3.34, \Delta = 0$	$T_1 = B \otimes B, T_2 = A \otimes C, T_3 = 4.0 \otimes T_2,$ $\Delta = T_1 \ominus T_3, \Delta == 0.0$	5
4	power.c (X=10,Y=-40) a-b-c-{d-e}^{40}-d-f-g-h	$RES = X^Y$ with $X = 10, Y = -40$	$W_1 = 0.0 \ominus Y, Z_1 = 1.0,$ $\{Z_{i+1} = Z_i * X, W_{i+1} = W_i - 1.0\}_{i=1..40},$ $Z_{42} = 1.0 \oslash Z_{41}, RES = Z_{42}$	84
5	power.c (X=10,Y=-350) a-b-c-{d-e}^{350}-d-f-g-h	$RES = X^Y$ with $X = 10, Y = -350$	$W_1 = 0.0 \ominus Y, Z_1 = 1.0,$ $\{Z_{i+1} = Z_i * X, W_{i+1} = W_i - 1.0\}_{i=1..350},$ $Z_{352} = 1.0 \oslash Z_{351}, RES = Z_{352}$	704
	Program	**Path condition over $\mathbb{R}$**	**Normalized FPSE path conditions**	**#**
6	foo1.c	$X > 0, X + 10^{12} = 10^{12}$	$X > 0.0, T_1 = X \oplus 1.0e12, T_1 = 1.0e12$	3
7	foo2.c	$X < 10^4, X + 10^{12} > 10^{12}$	$X < 10000.0, T_1 = X \oplus 1.0e12, T_1 > 1.0e12$	3
8	howden.c	$A * B + 2 > 100, 48 - A * B > 0$	$T_1 = A \otimes B, X_1 = T_1 \oplus 2.0, X_1 > 100.0,$ $X_2 = 100.0 \ominus X_1, X_3 = X_2 \ominus 50.0, X_3 > 50.0$	6
9	power.c (X,Y unknown) a-b-c-d-f-g-h	$Y < 0, Y \geq 0$	$Y < 0.0, W = \ominus Y, W \leq 0.0$	3
10	power.c (X,Y unknown) a-b-c-{d-e}^{40}-d-f-g-h	$Y < 0, Y < -39, Y \geq -40$	$Y < 0.0, W_1 = 0.0 \ominus Y,$ $\{W_i > 0.0, W_{i+1} = W_i - 1.0\}_{i=1..40}, W_{41} \leq 0.0$	83
11	power.c (X,Y unknown) a-b-c-{d-e}^{350}-d-f-g-h	$Y < 0, Y < -349, Y \geq -350$	$Y < 0.0, W_1 = 0.0 \ominus Y,$ $\{W_i > 0.0, W_{i+1} = W_i - 1.0\}_{i=1..350}, W_{351} \leq 0.0$	703

Results. In all the cases, the CPU time required to get the results with any of the four solvers (FPSE, IC, clpr, clpq) is less than a few seconds, so it is not shown. The first column contains the expected results computed either by executions of the C program or by manual analysis. In both cases, we provide the results over the singles and the doubles. Binary floating-point numbers are represented by decimal constants, noted with 16 decimals. The second column contains the results computed by the solvers over the reals and the rationals (IC, clpr and clpq). These solvers do not use single-format floating-point numbers, hence only the results over the double-format or the rationals is given. The last column contains the results computed by FPSE over both formats. Note that for any of the solvers (including FPSE), the labelling process has not been triggered and the results that are shown are obtained just after the constraint propagation step. Note that, as Eclipse IC is based on interval propagation, interval bounds are only changed if the absolute and relative changes of the bound exceed a given propagation threshold, which

is set to 1.0e-8.

Table 4

First experimental results

	Expected	with Eclipse	with FPSE
1	single:X = -1.0000000031710769e-30 double:X = -1.0000000000000001e-30	IC: X ∈]-1.0e-30, 140737488355328[clpr: X = 0.0 clpq: X = 1/999999999999998791471364833 28	single:X = -1.0000000031710769e-30 double:X = -1.0000000000000001e-30
2	single:Δ = 0.029199660219726562 double:Δ = 0.029200000000001225	IC: Δ ∈]0.029199999999997672, 0.029200000000001225[clpr: Δ = 0.029200000000001152 clpq: Δ = 73/2500 = 0.0292	single:Δ = 0.029199660219726562 double:Δ = 0.029200000000001225
3	single:C = 2.2859835624694824 double:C = 2.285983606557770	IC: C ∈]2.2859836065573766, 2.2859839065573771[clpr: C = 2.28598360657377 clpq: C = 27889/12200	single:C ∈]2.2859833240509033, 2.2859835624694824[double:C ∈]2.2859836065573766, 2.285983606557370[
4	single:RES = +0.0 double:RES = 1.0000000000000001e-40	IC: RES ∈]9.999999999999871e-41, 1.0000000000000016e-40[clpr: RES = 1.0e-40 clpq: RES = 10^{-40}	single: RES = +0.0 double:RES = 1.0000000000000001e-40
5	single:RES = +0.0 double: RES = +0.0	IC: RES ∈]0.0, 5.5626846462680 1e-309[clpr: RES = 1.0e-350 clpq: RES = 10^{-350}	single: RES = +0.0 double: RES = +0.0
	Expected	**with Eclipse**	**with FPSE**
6	single:X ∈]1.4012984643248171e-45, 3.2767998046875000e+04[double:X ∈]4.9406564584124654e-324, 6.1035156250000000e-05[	IC: infeasible path clpr: infeasible path clpq: infeasible path	single:X ∈]1.4012984643248171e-45, 3.2768000000000000e+04[double:X ∈]4.9406564584124654e-324, 6.1035156250000000e-05[
7	single:infeasible path double:X ∈]6.1035156250000000e-05, 9.9999999999999982e+03[	IC: X ∈]0.0, 10000.0[clpr: -0.0 < X < 10000.0 clpq: 0 < X < 10000	single:infeasible path double:X ∈]6.1035156250000000e-05, 9.9999999999999982e+03[
8	single:double:infeasible path	IC: infeasible path clpr: -48.0 + B*A < 0.0, 98.0 - B*A < 0.0 clpq: -48 + B*A < 0, 98 - B*A < 0	single:double:infeasible path
9	single:double:infeasible path	ic,clpr,clpq: infeasible path	single:double:infeasible path
10	single:Y ∈]-4.0e01, -39.000003814697265625[double:Y ∈]-4.0e01, -39.000000000000007105[	IC: Y ∈]-40.0,-39.0[clpr: -40.0 ≤ Y < -39.0 clpq: -40 ≤ Y < -39	single:Y ∈]-4.0e01, -39.0[double:Y ∈]-4.0e01, -39.0[
11	single:Y ∈]-350.0, -349.000030517578125[double:Y ∈]-350.0, -349.000000000000005684[	IC: Y ∈]-350.0,-349.0[clpr: -350.0 ≤ Y < -349.0 clpq: -350 ≤ Y < -349	single:Y ∈]-350.0, -349.0[double:Y ∈]-350.0, -349.0[

`Analysis.` First examples illustrate that the four evaluators may produce distinct results. In example 1, the results computed by both *clpr* and *clpq* are incorrect not only w.r.t. the expected result over the floats (first column) but also over the expected solutions over the reals (*i.e.* $+1.0e-30$). The library *IC* provides a correct but useless result over the reals as the superior bound of the

computed interval is greater than 10^{14}. As expected, FPSE provides the result strictly conforming to the evaluation of the program over the floating-point numbers (single and double), without any overestimation. Examples 2 and 3 show that even when expressions are not targeted to exemplify floating-point computation problems (`g2.c` computes the roots of the second order equation), the results given by the three solvers over the reals and the rationals (*IC, clpr, clpr*) do not conform to the ones computed by program executions. In example 3, FPSE returns an interval of 2 floating point values (in both cases) but only one of them satisfy the symbolic expression. Examples 4 and 5 show situations where floating-point numbers are flushed to zero by the computations, leading to a divergence with the computations over the reals (the program returns $+0.0$ instead of a strict positive quantity). FPSE provides the expected result as $1.0 \oslash +INF$ results in $+0.0$. Example 6 and 7 have already been discussed in the introduction of the paper. Examples 8 and 9 demonstrate path infeasibility. In example 8, both *clpr* and *clpq* return an unsolved non-linear constraint system. Solvers based on interval propagation (IC,FPSE) are not restricted to deal with linear constraints hence path infeasibility is shown. In example 9, all the four solvers provide the expected result. Finally, examples 10 and 11 illustrate the capacity of the solvers to deal with a realistic number of constraints, even when inverse projections are involved. In examples 8,9,10,11, IC and FPSE return the same (possibly overestimated) correct results at the end of the constraint propagation step, but only FPSE is trustworthy over the floating-point numbers.

To conclude, these experiments demonstrate that the proposed approach is suitable to deal efficiently with small-sized C floating-point computations. Of course, the set of experiments is too restricted to easily extrapolate the results

to larger computations but this work is a first attempt to address the problem of floating-point computations in symbolic execution.

9 Further work

In this paper, a new symbolic execution framework able to handle correctly IEEE-754 compliant floating-point computations has been introduced. The definitions of correct and efficient projection functions for solving normalized symbolic expressions have been given. Handling other rounding modes than the to-the-nearest number appears as being a tedious but not difficult extension of the proposed framework. In the same spirit, handling the square root function is straightforward: this function is included in the IEEE-754 standard and is correctly rounded. Dealing with extended formats appears to be an interesting extension as computations require more and more precision. This extension probably requires using multiple-precision floating-point numbers, as exploited in some computer algebra systems. The most difficult extension concerns the transcendental functions as there is nothing to guarantee that the computation is correctly rounded in these cases. This problem known as the table maker dilemma problem is likely to be the more prospective part of future work on this topic.

Acknowledgements

We are very grateful to Andy King for its careful reading of the paper and we wish to thank Michel Rueher for fruitful discussions on this work.

References

[1] King, J.C., "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[2] Goldberg, A. and Wang, T. and Zimmerman, D., "Applications of feasible path analysis to program testing", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WN, August 1994, pp. 80–92.

[3] Jasper, R. and Brennan, M. and Williamson, K. and Zimmerman, D., "Test data generation and feasible path analysis", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WN, August 1994, pp. 95–107.

[4] Weyuker, E., "Translatability and decidability questions for restricted classes of program schemas", *SIAM Journal of Computing*, vol. 8, no. 4, pp. 587–598, November 1979.

[5] Clarke, L., "A system to generate test data and symbolically execute programs", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, September 1976.

[6] Howden, W., "Reliability of the path analysis testing strategy", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 208–214, September 1976.

[7] Boyer, R. and Elspas, B. and Levitt, K., "SELECT - a formal system for testing and debugging programs by symbolic execution", *SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, June 1975.

[8] Ramamoorthy, C. and Ho, S. and Chen, W., "On the automated generation of program test data", *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 293–300, December 1976.

[9] Bicevskis, J. and Borzovs, J. and Straujums, U. and Zarins, A. and Miller, E., "SMOTL - a system to construct samples for data processing program debugging", *IEEE Transactions on Software Engineering*, vol. 5, no. 1, pp. 60–66, January 1979.

[10] Coward, D. and Ince, D., *The Symbolic Execution of Software - The SYM-BOL System*, Chapman & Hall, London, UK, 1995.

[11] Meudec, C., "ATGen: automatic test data generation using constraint logic programming and symbolic execution", *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 81–96, June 2001.

[12] DeMillo, R.A. and Offut, J.A., "Experimental results from an automatic test case generator", *ACM Transactions on Software Engineering Methodology*, vol. 2, no. 2, pp. 109–127, April 1993.

[13] Coen-Porisini, A. and de Paoli, F. and Ghezzi, C. and Mandrioli, D., "Software specialization via symbolic execution", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 884–899, September 1991.

[14] Fahringer, T. and Scholz, B., "A unified symbolic evaluation framework for parallelizing compilers", *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 11, pp. 1105–1125, November 2000.

[15] Coen-Porisini, A. and Denaro, G. and Ghezzi, C. and Pezze, M., "Using symbolic execution for verifying safety-critical systems", in *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*, Vienna, Austria, September 2001, ACM, pp. 142–150.

[16] Chen, T.Y. and Tse, T.H. and Zhou, Z., "Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing", in *Proceedings*

of the International Symposium on Software Testing and Analysis (ISSTA'02), Roma, Italy, July 2002, pp. 191–195.

[17] Goldberg, D., "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, March 1991.

[18] IEEE-754, "Standard for binary floating-point arithmetic", *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 9–25, February 1985.

[19] Brisset, P. and Sakkout, H. and Fruhwirth, T. and Gervet, C. and Harvey, et al., *ECLiPSe Constraint Library Manual*, International Computers Limited and Imperial College London, UK, 2005, Release 5.8.

[20] Granvilliers, L., *RealPaver User's Manual : Solving Nonlinear Constraints by Interval Computations*, University of Nantes, FR, 2003, Release 0.3.

[21] Botella, B. and Taillibert, P., "Interlog : Constraint logic programming on numeric intervals", in *Third International Workshop on Software Engineering, Artificial Intelligence and Expert Systems*, Oberammergau, October 1993.

[22] Lhomme, O. and Gotlieb, A. and Rueher, M. and Taillibert, P., "Boosting the interval narrowing algorithm", in *Proceeedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, Bonn, September 1996, MIT Press, pp. 378–392.

[23] Michel, C., "Exact projection functions for floating point number constraints", in *Proceedings of seventh AIMA Symposium*, Fort Lauderdale, FL, USA, 2002.

[24] Miller, W. and Spooner, D., "Automatic generation of floating-point test data", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, September 1976.

[25] Gotlieb, A. and Botella, B. and Rueher, M., "Automatic test data generation using constraint solving techniques", in *Proceedings of the International*

Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, FL, USA, March 1998, pp. 53–62.

[26] Gotlieb, A. and Botella, B. and Rueher, M., "A clp framework for computing structural test data", in *Proceedings of Computational Logic (CL'2000)*, London, UK, July 2000, LNAI 1891, pp. 399–413.

[27] Holzbaur, C., *OEFAI clp(q,r) Manual Rev. 1.3.2*, Austrian Research Institute for Artificial Intelligence, Vienna, AU, 1995, TR-95-09.

[28] Muchnick, S. and Jones, N., *Program Flow Analysis: Theory and Applications – Chapter 9 : L. Clarke, D. Richardson*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[29] Coen-Porisini, A. and de Paoli, F., "Array representation in symbolic execution", *Computer Languages*, vol. 18, no. 3, pp. 197–216, 1993.

[30] Dillon, E. and Meudec, C., "Automatic test data generation from embedded c code", in *Proceedings of SAFECOMP'04*, Potsdam, Germany, September 2004, Springer Verlag, LNCS 3219, pp. 180–194.

[31] Benhamou, F. and McAllester, D. and van Hentenryck, P., "CLP(Intervals) revisited", in *Proceedings of the 1994 International Symposium on Logic Programming (ILPS'94)*, Ithaca, New York, November 1994, pp. 124–138, MIT Press.

[32] Benhamou, F. and Older, W., "Applying interval arithmetic to real, integer and boolean constraints", *Journal of Logic Programming*, vol. 32, no. 1, pp. 1–24, July 1997.

[33] Moore, R.A., *Interval Analysis*, Prentice Hall, New Jersey, 1966.

[34] Cleary, J.G., "Logical arithmetic", *Future Computing Systems*, vol. 2, no. 2, pp. 125–149, 1987.

[35] Older, W. and Vellino, A., "Extending prolog with constraints arithmetic on reals intervals", in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*. 1990, IEEE Computer Society Press.

[36] Hickey, T.J. and Ju, Q. and van Emden, M.H., "Interval arithmetic: From principles to implementation", *Journal of ACM*, vol. 48, no. 5, pp. 1038–1068, September 2001.

[37] Hauser, J.R., "Handling floating-point exceptions in numeric programs", *ACM Transactions on Programming Language and Systems*, vol. 18, no. 2, pp. 139–174, March 1996.

[38] Michel, C. and Rueher, M. and Lebbah, Y., "Solving constraints over floating-point numbers", in *Proceedings of Principles and Practices of Constraint Programming (CP'01)*, Paphos, Cyprus, November 2001, Springer Verlag, LNCS 2239, pp. 524–538.

[39] DeMillo, R.A. and Offut, J.A., "Constraint-based automatic test data generation", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[40] Howden, W., "Validation of scientific programs", *ACM Computing Surveys*, vol. 14, no. 2, pp. 193–227, June 1982.

Appendix

This appendix contains the tables used in direct and inverse projections when infinities are involved in the computations.

Table 5

Value of r in direct $proj(r, r = a \ominus b)$

$a \backslash b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$\perp$	$-INF$	$-INF$	$-INF$	$-INF$
-0.0	$+INF$	$+0.0$	-0.0	Nv	$-INF$
$+0.0$	$+INF$	$+0.0$	$+0.0$	Nv	$-INF$
Nv	$+INF$	Nv	Nv	$Nv \cup \{\pm INF, +0.0\}$	$-INF$
$+INF$	$+INF$	$+INF$	$+INF$	$+INF$	$\perp$

Table 6

Value of r in direct $proj(r, r = a \otimes b)$

$a \backslash b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$+INF$	$\perp$	$\perp$	$-INF$	$-INF$
-0.0	$\perp$	$+0.0$	-0.0	$\{\pm 0.0\}$	$\perp$
$+0.0$	$\perp$	-0.0	$+0.0$	$\{\pm 0.0\}$	$\perp$
Nv	$\{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	$Nv \cup \{\pm 0.0, \pm INF\}$	$\{\pm INF\}$
$+INF$	$-INF$	$\perp$	$\perp$	$+INF$	$+INF$

Table 7

Value of r in direct $proj(r, r = a \oslash b)$

$a \backslash b$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$\perp$	$+INF$	$-INF$	$\{-INF, +INF\}$	$\perp$
-0.0	$+0.0$	$\perp$	$\perp$	$\{\pm 0.0\}$	-0.0
$+0.0$	-0.0	$\perp$	$\perp$	$\{\pm 0.0\}$	$+0.0$
Nv	$\{\pm 0.0\}$	$\{\pm INF\}$	$\{\pm INF\}$	$Nv \cup \{\pm 0.0, \pm INF\}$	$\{\pm 0.0\}$
$+INF$	$\perp$	$-INF$	$+INF$	$\{-INF, +INF\}$	$\perp$

Table 8

Value of a in first inverse $proj(a, r = a \ominus b)$

$b \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$\perp$	$\perp$	$\perp$	$\perp$	$Nv \cup \{+INF, \pm 0.0\}$
-0.0	$-INF$	$\perp$	$\{\pm 0.0\}$	Nv	$+INF$
$+0.0$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
Nv	$Nv \cup \{-INF\}$	$\perp$	Nv	$Nv \cup \{\pm 0.0\}$	$Nv \cup \{+INF\}$
$+INF$	$Nv \cup \{-INF, \pm 0.0\}$	$\perp$	$\perp$	$\perp$	$\perp$

Table 9

Value of a in first inverse $proj(a, r = a \otimes b)$

$b \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$Nv \cup \{+INF\}$	$\perp$	$\perp$	$\perp$	$Nv \cup \{-INF\}$
-0.0	$\perp$	$Nv \cup \{+0.0\}$	$Nv \cup \{-0.0\}$	$\perp$	$\perp$
$+0.0$	$\perp$	$Nv \cup \{-0.0\}$	$Nv \cup \{+0.0\}$	$\perp$	$\perp$
Nv	$Nv \cup \{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	Nv	$Nv \cup \{\pm INF\}$
$+INF$	$Nv \cup \{-INF\}$	$\perp$	$\perp$	$\perp$	$Nv \cup \{+INF\}$

Table 10

Value of a in first inverse $proj(a, r = a \oslash b)$

$b \backslash r$	$-INF$	-0.0	$+0.0$	Nv	$+INF$
$-INF$	$\perp$	$Nv \cup \{+0.0\}$	$Nv \cup \{-0.0\}$	$\perp$	$\perp$
-0.0	$Nv \cup \{+INF\}$	$\perp$	$\perp$	$\perp$	$Nv \cup \{-INF\}$
$+0.0$	$Nv \cup \{-INF\}$	$\perp$	$\perp$	$\perp$	$Nv \cup \{+INF\}$
$+INF$	$\perp$	$Nv \cup \{-0.0\}$	$Nv \cup \{+0.0\}$	$\perp$	$\perp$
Nv	$\{\pm INF\}$	$\{\pm 0.0\}$	$\{\pm 0.0\}$	Nv	$\{\pm INF\}$

Travaux connexes et portée de l'article

En parallèle des travaux entrepris dans les projets INKA et V3F, des recherches ont été menées pour la prise en compte des calculs flottants en Interprétation Abstraite [Goubault 01] avec le domaine abstrait des intervalles et celui des polyèdres convexes [Miné 04]. Ces travaux ont donné lieu à deux outils essentiels pour l'analyse statique de programmes C contenant des calculs flottants: Fluctuat [Delmas 09] et Astrée [Cousot 05]. L'objectif consistant à étudier la stabilité numérique des calculs flottants a également été poursuivi plus récemment par les travaux de Tang *et al.* [Tang 10]. Néanmoins, la prise en compte des flottants dans les calculs d'intervalles utilisés en Interprétation Abstraite ne partage pas la problématique de la résolution de contraintes sur les flottants. En effet, ils sont basés sur une interprétation en avant des programmes C, ce qui ne nécessite pas la modélisation des projections indirectes sur les domaines abstraits utilisés pour les flottants. En s'appuyant sur une formalisation de la norme IEEE-754, il a néanmoins été proposé dans [Boldo 09] de certifier les calculs abstraits flottants en utilisant le système de preuve Coq.

En génération automatique de test où la modélisation des projections indirectes est nécessaire, deux approches principales ont été proposées. La première est basée sur l'essai de valeurs et la réfutation [Miller 76, Lakhotia 10] et peut-être comprise comme une variation de techniques de *Recherche Locale* [Arcuri 09]. La seconde est la modélisation des erreurs d'arrondis directement au coeur de la procédure de décision utilisée. Bien qu'il ait été noté que les calculs flottants ne sont pas systématiquement impliqués dans le flot de contrôle et peuvent donc être ignorés dans certains cas pour la génération automatique de données de test [Godefroid 10], plusieurs approches récentes visent à traiter correctement les calculs flottants au sein des résolveurs SMT (*Satisfiability Modulo Theory*). L'approche qui prend en compte les flottants en exécution symbolique, a longtemps consisté à les traiter comme des rationnels ou des réels, en ignorant les erreurs d'arrondi. Les solveurs SMT tels que Yices ou Z3 [de Moura 08a] utilisent des procédures de décision basées sur la Programmation Linéaire (i.e., simplexe, élimination de Fourier-Motzkin, logique de différence) pour les contraintes linéaires. Sachant que ces procédures sont elles-mêmes implantées avec les flottants, leur résultat ne doit être interprété qu'avec beaucoup de circonspection. Néanmoins, un effort récent supporté par Microsoft[2] a été lancé pour proposer une théorie normative des calculs sur les flottants utilisable dans les solveurs SMT. Une autre approche a visé à combiner des sur-approximations et sous-approximations de calculs flottants dans un calcul itératif, pour approcher les résultats de la résolution d'expressions sur les flottants [Brillout 09]. Dans le cadre des travaux sur la résolution de contraintes sur les flottants [Michel 01, Michel 02, Botella 06], une approche pour améliorer les capacités de filtrage des contraintes consiste à tirer parti des propriétés numériques des calculs flottants. Pour les contraintes linéaires, cela conduit à une technique de relaxation linéaire sur les réels [Mohammed Said Belaid 10], tandis que pour les approches à base de propagation d'intervalles, Marre et Michel ont proposé

[2] `http://www.cprover.org/SMT-LIB-Float/`

dans [Marre 10] d'exploiter la représentation des flottants dans les algorithmes de filtrage de l'addition et la soustraction flottante. Tout récemment, nous avons proposé de généraliser ce travail en le reformulant, et en proposant des algorithmes de filtrage pour la multiplication et la division flottante [Carlier 11b].

Ce chapitre clos la seconde partie du mémoire sur quelques développements auxquels nous avons contribués dans le domaine du test à base de contraintes. Ces développements ont eu des applications dans le domaine du test logiciel que nous évoquons dans la patrie suivante.

Part III

Applications

Chapter 8

Génération de tests pour Java Card

La promesse du déploiement massif de terminaux mobiles intégrant des capacités d'exécutions d'applications embarquées est désormais une réalité (e.g., smartphones, mobiles Java, carte à puce Java). Ces terminaux, munis d'une carte à puce ouverte, s'appuient le plus souvent sur des versions spécialisées de la plateforme Java telles que J2ME ou Java Card. Les applications logicielles exécutables sur ces terminaux mobiles doivent être validées et certifiées par les opérateurs afin d'offrir des garanties de sécurité et de sûreté de fonctionnement. Ces phases de validation et de certification sont aujourd'hui reconnues comme essentielles mais très coûteuses. En effet, elles reposent sur des techniques de test et d'analyse statique qui sont encore très artisanales. En particulier, les scenarii de test qui visent à détecter des comportements dangereux du terminal ou des défaillances, sont créés manuellement. Un des enjeux économiques de ce domaine consiste à mettre au point des techniques de tests permettant la validation et la certification des cartes la plus automatisée possible.

Ce chapitre présente deux applications du test à base de contraintes, à la validation de la plateforme Java Card, qui est le dialecte Java dédié à la carte à puce. Nous nous sommes intéressés à Java Card dans le cadre du projet RNTL CASTLES (2003-2006) qui visait à bâtir des outils pour la certification de la machine virtuelle Java Card ("Java Card Virtual Machine", JCVM). Le premier des deux articles de ce chapitre propose l'utilisation des *"Constraint Handling Rules"* pour générer des tests pour chaque instruction de la JCVM. Partant d'une formalisation Jakarta (un dialecte de Coq) de la JCVM, un modèle à contraintes sous forme de règles CHR est automatiquement généré pour chaque instruction et ensuite résolu pour instancier un état abstrait de la machine virtuelle. Cette application des CHR a été reconnue comme étant pertinente [Sneyers 10] et plusieurs travaux concernant la génération automatique de tests avec des CHRs ont tiré parti des résultats de ce papier. En particulier, on peut mentionner les travaux de Degrave *et al.* [Degrave 09] et Gerlich [Gerlich 10] qui utilisent les CHRs pour générer automatique des cas de test

pour les programmes impératifs.

S.D. Gouraud and A. Gotlieb. *Using chrs to generate test cases for the JCVM*. **In Eighth International Symposium on Practical Aspects of Declarative Languages (PADL'06), Charleston, South Carolina, January 2006. LNCS 3819.**

Using CHRs to generate functional test cases for the Java Card Virtual Machine*

Sandrine-Dominique Gouraud and Arnaud Gotlieb

IRISA/CNRS UMR 6074,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, FRANCE
Phone: +33 (0)2 99 84 75 76 – Fax: +33 (0) 2 99 84 71 71
gouraud@lri.fr, gotlieb@irisa.fr

Abstract. Automated functional testing consists in deriving test cases from the specification model of a program to detect faults within an implementation. In our work, we investigate using Constraint Handling Rules (CHRs) to automate the test cases generation process of functional testing. Our case study is a formal model of the Java Card Virtual Machine (JCVM) written in a sub-language of the Coq proof assistant. In this paper we define an automated translation from this formal model into CHRs and propose to generate test cases for each bytecode definition of the JCVM. The originality of our approach resides in the use of CHRs to faithfully model the formally specified operational semantics of the JCVM. The approach has been implemented in Eclipse Prolog and a full set of test cases have been generated for testing the JCVM.

Keywords: CHR, Software testing, Java Card Virtual Machine.

1 Introduction

The increasing complexity of computer programs ensures that automated software testing will continue to play a prevalent role in software validation. In this context, automated functional testing consists in 1) generating test cases from a specification model, 2) executing an implementation using the generated test cases and then 3) checking the computed results with the help of an oracle. In automated functional testing, oracles are generated from the model to provide the expected results. Several models have been used to generate test cases: algebraic specifications [1], B machineries [2] or finite state machines [3], just to name a few.
In our work, we investigate using Constraint Handling Rules (CHRs) to automate the test cases and oracles generation process of functional testing. Our specification model is written in a sub-language of Coq: the Jakarta Specification Language (JSL) [4]. Coq is the INRIA's proof assistant [5] based on the calculus of inductive constructions that allows to mechanically prove high-order theorems. Recently, Coq and JSL were used to derive certified Byte Code Verifiers by abstraction from the specification of a Java Card Virtual Machine [4, 6]. The Java Card Virtual Machine (JCVM) carries out all the instructions (or bytecodes) supported by Java Card (new, push, pop, invokestatic, invokevirtual, etc.). In this paper, we present how to generate test cases and oracles for each JSL byte code specification. Our idea is to benefit from the high declarativity of CHRs to express the test purpose as well as the JSL specification rules into a single framework. Then, by using traditional CHR propagation and labelling, we generate test cases and oracles as solutions of the underlying constraint system. The approach has been implemented with the CHR library of Eclipse Prolog [7] and a full set of test cases have been generated for testing the JCVM.
This paper is organised as follows: Section 2 introduces JSL and its execution model; Section 3 recalls some background on CHRs; Section 4 introduces the translation rules used to convert a formal specification written in JSL into CHRs; Section 5 presents our algorithm to generate functional test cases and oracles for testing an implementation of the JCVM; Section 6 describes some related works, and finally Section 7 concludes the paper with some research perspectives.

* This work is supported by the Réseau National des Technologies Logicielles as part of the CASTLES project (www-sop.inria.fr/everest/projects/castles/). This project aims at defining a certification environment for the JavaCard platform. The project involves two academic partners: the Everest and Lande teams of INRIA and two industrial partners: Oberthur Card Systems and Alliance Qualit Logicielle.

2 The Jakarta Specification Language

The Jakarta Specification Language (JSL), as introduced in [8], is a first order language with a polymorphic type system. JSL functions are formally defined with conditional rewriting rules.

2.1 Syntax

JSL expressions are first order terms with equality (==), built from term variables and from constant symbols. A constant symbol is either a constructor symbol introduced by data types definitions or a function symbol introduced by function definitions.
Let $\mathcal{C}$ be a set of constructor symbols, $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ be a set of term variables. The JSL expressions set is the term set $\mathcal{E}$ defined by: $\mathcal{E}::= \ \mathcal{V}|\mathcal{E} == \mathcal{E}|\mathcal{C}\mathcal{E}^*|\mathcal{F}\mathcal{E}^*$. Let var be the function defined on $\mathcal{E} \rightarrow \mathcal{V}^*$ which returns the set of variables of a JSL expression.
Each function symbol is defined by a set of conditional rewriting rules. This unusual format for rewriting is close to functional language with pattern-matching and proof assistant. These (oriented) conditional rewriting rules are of the form $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ where:

- $g = f v_1 \ldots v_m$ where $\forall i, v_i \in \mathcal{V}$ and $\forall i, j, v_i \neq v_j$
- l_i is either a variable or a function which does not introduce new variables: for $1 \leq i \leq n$, $var(l_i) \subseteq var(g) \cup var(r_1) \cup \ldots \cup var(r_{i-1})$
- r_i should be a value called *pattern* (built from variables and constructors), should contain only fresh variables and should be linear[1]:

[1] All the variables are required to be distinct

for $1 \leq i, j \leq n$ and $i \neq j$, $var(r_i) \cap var(g) = \emptyset$ and $var(r_i) \cap var(r_j) = \emptyset$

– d is an expression and $var(d) \subseteq var(g) \cup var(r_1) \ldots \cup var(r_n)$

The rule means if for all i, l_i can be rewritten into r_i then g is rewritten into d. Thereafter, these rules are called JSL rules. JSL allows the definition of partial or non-deterministic functions.

Example 1 (JSL def. of **plus** *extracted from the JCVM formal model).*
$data\, nat\ =\ 0 \mid S\, nat.$
$function$ **plus** :=
$\langle plus\_r1 \rangle \quad n \rightarrow 0 \quad \Rightarrow (plus\, n\, m) \rightarrow m;$
$\langle plus\_r2 \rangle \quad n \rightarrow (S\, p) \Rightarrow (plus\, n\, m) \rightarrow (S\, (plus\, p\, m)).$

2.2 Execution model of JSL

Let $e_{|p}$ denote the subterm of e at position p then expression $e[p \leftarrow d]$ denotes the term e where $e_{|p}$ is replaced by term d.
Let $\mathcal{R}$ be a set of rewriting rules, then an expression e is rewritten into e' if there exists a rule $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ in $\mathcal{R}$, a position p and a substitution θ such as:

– $e_{|p} = \theta g$ and $e' = e[p \leftarrow \theta d]$
– $\{\theta l_i \rightarrow^* \theta r_i\}_{\forall 1 \leq i \leq n}$ where $\rightarrow^*$ is the transitive cloture of $\rightarrow$

Note that nothing prevents JSL specifications to be non-terminating or non-confluent. However, the formal model of the JCVM we are using as a case study has been proved terminating and confluent within the Coq proof assistant [4, 6].

Example 2 (Rewriting of $(plus\ 0\ (plus(S\ 0)\ 0)))$*).*
$(plus\ 0\ (plus\ (S\ 0)\ 0)) \rightarrow_{r1} (plus\ (S\ 0)\ 0) \rightarrow_{r2} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)$

3 Background on Constraint Handling Rules

This section is inspired of Thom Frühwirth's survey and book [9, 10]. The Constraint Handling Rules (CHRs) language is a committed-choice language, which consists of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. This language extends a host language with constraint solving capabilities. Implementations of CHRs are available in Eclipse Prolog [7], Sicstus Prolog, HAL [11], etc.

3.1 Syntax

The CHR language is based on **simplification** where constraints are replaced by simpler ones while logical equivalence is preserved and **propagation** where new constraints which are logically redundant are added to cause further simplification. A constraint is either a built-in (predefined) first-order predicate or a CHR (user-defined) constraint defined by a finite set of CHR rules. Simplification rules are of the form `H <=> G | B` and propagation rules are of the form `H ==> G | B` where H denotes a possibly multi-head CHR constraint, the guard G is a conjunction of constraints and the body B is a conjunction of built-in and CHR constraints.

Each time a CHR constraint is woken, its guard must either succeed or fail. If the guard succeeds, one commits to it and then the body is executed. Constraints in the guards are usually restricted to be built-in constraints. When other constraints are used in the guards (called *deep guards*), special attention must be paid to the way guards are evaluated. Section 4.2 discusses the use of *deep guards* in our framework.

Example 3 (CHRs that can be used to define the `plus` *constraint).*

```
R1 @ plus(A,B,R) <=> A=0    | R=B.
R2 @ plus(A,B,R) <=> A=s(C) | plus(C,B,D), R=s(D).
C  @ plus(A,B,R) ==> plus(B,A,R).
```

The construction `...@` gives names to CHRs.

3.2 Semantics

Given a constraint theory (CT) (with true, false and an equality constraint =) which determines the meaning of built-in constraints, the declarative interpretation of a CHR program is given by a conjunction of universally quantified logical formula. There is a formula for each rule.
If $\bar{x}$ denotes the variables occurring in the head H and $\bar{y}$ (resp. $\bar{z}$) the variables occurring in the guard (resp. body) of the rule, then

– a simplification CHR is interpreted as $\forall \bar{x}(\exists \bar{y} G \rightarrow (H \leftrightarrow \exists \bar{z} B))$
– a propagation CHR is interpreted as $\forall \bar{x}(\exists \bar{y} G \rightarrow (H \rightarrow \exists \bar{z} B))$

The operational semantics of CHR programs is given by a transition system where a state $< G, C >$ consists of two components: the goal store G and the constraint store C. An initial state is of the form $< G, true >$. A final state $< G, C >$ is successful when no transition is applicable whereas it is failed when $C = false$ (the constraint store is contradictory).

Solve If C is a built-in constraint and $CT \models (C \wedge D) \leftrightarrow D'$
Then $< C \wedge G, D > \mapsto < G, D' >$
Simplify If $F <=> D|H$ and $CT \models \forall (C \rightarrow \exists \bar{x}(F = E \wedge D)$
Then $< E \wedge G, C > \mapsto < H \wedge G, (F = E) \wedge D \wedge C >$
Propagate If $F => D|H$ and $CT \models \forall (C \rightarrow \exists \bar{x}(F = E \wedge D)$
Then $< E \wedge G, C > \mapsto < E \wedge H \wedge G, (F = E) \wedge D \wedge C >$

Rules are applied fairly (every rule that is applicable is applied eventually). Propagation rule is applied at most once on the same constraints in order to avoid trivial non-termination. However, CHR programs can be non-confluent and non-terminating.

Example 4 (Several examples of the CHR solving process).

```
                plus(s(0),s(0),R)
↦Simplify_R2    plus(0,s(0),R1), R=s(R1)
↦Simplify_R1    R1=s(0), R=s(R1)
↦Solve          R=s(s(0))
```

The following example exploits the propagation rule of **plus**. Without this rule, the term `plus(M,s(0),s(s(0)))` would be delayed.

```
                plus(M,s(0),s(s(0)))
↦Propagate_C    plus(M,s(0),s(s(0))), plus(s(0),M,s(s(0)))
↦Simplify_R2    plus(M,s(0),s(s(0))), plus(0,M,s(0))
↦Simplify_R1    plus(M,s(0),s(s(0))), M=s(0)
↦Solve          plus(s(0),s(0),s(s(0))), M=s(0)
↦Simplify_R2    plus(0,s(0),s(0)), M=s(0)
↦Simplify_R1    s(0)=s(0), M=s(0)
↦Solve          M=s(0)
```

The following example shows the deduction of a relation ($M = N$):

```
                plus(M,0,N)
↦Propagate_C    plus(M,0,N), plus(0,M,N)
↦Simplify_R1    plus(M,0,N), M=N
↦Solve          plus(M,0,M), M=N
```

4 JSL to CHR translation method

Our approach is based on the syntactical translation of JSL specifications into CHRs. The translation method is described under the form of judgements.

4.1 Translation method

There are three kinds of judgements: judgements for JSL expressions, judgements for JSL rewriting rules (main operator $\rightarrow$) and judgements for JSL functions (main operator $\Rightarrow$).
The judgement $e \rightsquigarrow \mathtt{t} \lhd \{\mathtt{C}\}$ states that JSL expression e is translated into term $\mathtt{t}$ under the conjunction of constraints $\mathtt{C}$.

$$\frac{variable(v)}{v \rightsquigarrow \mathtt{v} \lhd \{\mathtt{true}\}} \qquad \frac{constant(c)}{c \rightsquigarrow \mathtt{c} \lhd \{\mathtt{true}\}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\}}{c\, e_1 \ldots e_n \rightsquigarrow \mathtt{c}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \lhd \{\mathtt{c_1}, \ldots, \mathtt{c_n}\}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\}}{f\, e_1 \ldots e_n \rightsquigarrow \mathtt{r} \lhd \{\mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{r})\}}$$

The judgement $(e \rightarrow p) \rightsquigarrow \{\mathtt{C}\}$ states that the JSL rewriting rule $e \rightarrow p$ is translated into the conjunction of constraints $\{\mathtt{C}\}$.

$$\frac{}{(v \rightarrow p) \rightsquigarrow \{\mathtt{v} = \mathtt{p}\}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\} \quad p \rightsquigarrow \mathtt{p} \lhd \{\mathtt{true}\}}{(f\, e_1 \ldots e_n \rightarrow p) \rightsquigarrow \{\mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{p})\}}$$

The judgement $(l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d) \rightsquigarrow \mathtt{g'} \Leftrightarrow \mathtt{guard}|\mathtt{body}$ states that the JSL function rule $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ is translated into the CHR $\mathtt{g'} \Leftrightarrow \mathtt{guard}|\mathtt{body}$ where $\mathtt{g'}$ is a CHR constraint associated to the expression g, $\mathtt{guard}$ is the conjunction of constraints corresponding to the translation of the rules $l_i \rightarrow r_i$, and $\mathtt{body}$ is a conjunction of constraints corresponding to the translation of the expression d.

$$\frac{l_1 \rightarrow r_1 \rightsquigarrow \mathtt{g_1} \quad \ldots \quad l_n \rightarrow r_n \rightsquigarrow \mathtt{g_n} \qquad e \rightsquigarrow \mathtt{t} \lhd \{\mathtt{B}\}}{(l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow f\, v_1 \ldots v_k \rightarrow e) \rightsquigarrow \mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_k}, \mathtt{r}) \Leftrightarrow \mathtt{g_1}, \ldots, \mathtt{g_n}|\mathtt{B}, \mathtt{r} = \mathtt{t}.}$$

Note that non-determinism, confluence and termination are preserved by the translation as the operational semantics of CHRs extends the execution model of JSL functions.

4.2 Deep guards

In the translation method, we considered that CHR guards could be built over prolog goals and CHR calls. This approach, which is referred to as deep guards, has received much attention by the past. See [9, 12] for a detailed presentation of deep guards. Smolka recalls in [13] that "deep guards constitute the central mechanism to combine processes and (encapsulated) search for problem-solving". Deep guards are used in several systems such as AKL, Eclipse Prolog [7, 9], Oz [12] or HAL [11]. Deep guards rely on how guard entailment is tested in conditional constraints and CHRs. Technically, a guard entailment test is called an "ask constraint" whereas a constraint added to the constraint store is called a "tell constraint" and both operations are clearly distinct. For example, if the constraint store contains $X = p(Z), Y = p(a)$ then a tell constraint $X = Y$ where $=$ denotes Prolog unification, will result in the store $X = p(a), Y = p(a), Z = a$ whereas the corresponding ask constraint will leave the store unchanged and will suspend until the constraint $Z = a$ would be entailed or disentailed.
The current approach to deal with deep guards that contain Prolog goals (but not CHR calls) consists in considering guards as tell constraints and checking at runtime that no guard variable is modified. This approach is based on the fact that the only way of constraining terms in the Herbrand Universe is unification ($=$) and that the corresponding ask constraint of unification is well-known: this is the "equality of terms" test ($==$). For example, if $X = Y$ is a tell constraint then $X == Y$ corresponds to its ask constraint. However, when Prolog goals are involved into the guards, the guard entailment test is no more decidable as non-terminating computations can arise. Note that CHR programs are not guaranteed to terminate (consider for example $p <=> true|p$). Even when non-terminating computations are avoided this approach can be very inefficient as possible long term computations in guards are executed every time a CHR constraint is woken. An approach for this problem consists in pre–computing the guard by executing the Prolog goal only once, and then testing entailment on the guard variables.
When CHRs are involved into the guards, the problem is more difficult as guards can set up constraints. In that case, considering guards as tell constraints is no longer correct as wrong deductions can be made. Our approach for this problem consists in suspending the guard entailment

test until it could be decided. More precisely, the guard entailment test is delayed until all the guard variables become instantiated[2]. At worst, this instantiation arises during the labelling process. Of course, this approach leads to fewer deductions at propagation time but it remains manageable when we have to deal with deep guards containing CHR calls.

4.3 Implementation of the translation method

We implemented the translation method into a library called `JSL2CHR.pl`. Given a file containing JSL definitions, the library builds an abstract syntax tree by using a Definite Clause Grammar of JSL, and then automatically produces equivalent CHR rules. The library was used on the JSL specifications of the JCVM, which is composed of 310 functions. As a result, 1537 CHRs were generated.

5 Tests generation for the JCVM

This section is devoted to the presentation of both the JCVM specification model and the test cases and oracle generation method. The experimental results we obtained by generating test cases for the JCVM are presented in Section 5.3.

5.1 The Java Card Virtual Machine

Unlike other smart cards, a Java Card includes a Java Virtual Machine implemented in its read-only memory part. The structure of a Java Card platform is given in Fig.1. It consists of several components, such as a runtime environment, an implementation of the Java Virtual Machine, the open and global platform applications, a set of packages implementing the standard SUN's Java Card API and a set of proprietary APIs. A Java Card program is called an applet and communicates with a card reader through APDU[3] buffers.
All the components of a Java Card platform must be thoroughly tested before the Card would be released. But, in this paper, we concentrate only on the JCVM functional testing process. In the formal model given in [14], the JCVM is a state machine described by a small-step semantics: each bytecode is formalised as a state transformer.

States modelling Each state contains all the elements manipulated by a program during its execution: values, objects and an execution environment for each called method. States are formalised as a record consisting of a heap (he) which contains the objects created during execution, a static heap (sh) which contains static fields of classes and a stack of frames (fr) which contain the execution environments of methods. States are tagged "Abnormal" if an exception (or an error) is raised, "Normal" otherwise.

[2] This solution is close to the traditional techniques of coroutining in Prolog as implemented by `freeze` or `delay` built-in predicates.

[3] Application Protocol Data Unit is an ISO-normalised communication format between the card and the off-card applications.

Bytecodes modelling The JCVM contains 185 distinct bytecodes which can be classified into the following classes[15]: arithmetic operations (`sadd`, `idiv`, `sshr`, ...), type verifications on objects (`instanceof`, ...), (conditional) branching (`ifcmp`, `goto`, ...), method calls (`invokestatic`, `invokevirtual`, ...), operations on local variables (`iload`, `sstore`, ...), operations on objects (`getfield`, `newarray`, ...), operations on operands stack (`ipush`, `pop`, ...) and flow modifiers (`sreturn`, `throw`, ...).
Most of the bytecodes have a similar execution scheme: to decompose the current state, to get components of the state, to perform tests in order to detect execution errors then to build the next state. In the JSL formal model of the JCVM, several bytecodes are specified with the similar JSL functions. They only distinguish by their type which is embodied in the JSL function definition as a parameter. As a result, the model contains only 45 distinct JSL functions associated to the bytecodes. Remaining functions are auxiliary functions that perform various computations. Some JSL functions calls other functions in their rewriting rules; this process is modelled by using deep guards in CHR, preserving so the operational semantics of the JCVM.

Example of a JSL bytecode specification As an example, consider the JSL specification of bytecode *push*: given a primary type t, a value x and a JCVM state st, *push* updates the operand stack of the first execution method environment in st by adding the value x of type t:
$function$ **push** :=
$\langle push\_r1 \rangle\ (stack\_f\ st) \rightarrow Nil$
$\quad \Rightarrow (push\ t\ x\ st) \rightarrow (abortCode\ State\_error\ st);$
$\langle push\_r2 \rangle\ (stack\_f\ st) \rightarrow (Cons\ h\ lf)$
$\quad \Rightarrow (push\ t\ x\ st) \rightarrow (update\_frame(result\_push\ t\ x\ h)\ st).$

push uses the auxiliary function **stack_f** that returns the stack of frames (environments for executing methods) of a given state.
$function$ **stack_f** :=
$\langle stack\_f\_r1 \rangle\ st \rightarrow (Jcvm\_state\ sh\ he\ fr) \Rightarrow (stack\_f\ st) \rightarrow fr.$

Example of CHR generated for a bytecode The following CHRs were produced by the library `JSL2CHR.pl`:

```
stack_f_r1 @ stack_f(St,R) <=> St=jcvm_state(Sh,He,Fr)
    | R=Fr.
push_r1 @ push(T,X,St,R) <=> stack_f(St,nil)
    | abortCode(state_error(St),Ra), R=Ra.
push_r2 @ push(T,X,St,R) <=> stack_f(St,cons(H,Lf))
    | result_push(T,X,H,Res), update_frame(Res,St,Ru), R=Ru.
```

In this example, the JSL function **stack_f** was translated into a CHR although it is only an accessor. As a consequence we get a deep guard in the definition of CHR `push`. This could be easily optimised by identifying the accessors into the JSL specification with the help of the user. However, we would like the approach to remain fully automated hence we did not realized this improvement and maintained the deep guards.

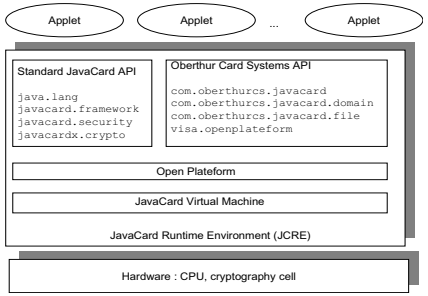


Fig. 1. A Java Card platform

5.2 Test cases and oracles generation method

Our approach is inspired of classical functional testing where test cases are generated according to some coverage criteria. We proposed to generate test cases that ensure each CHR would be covered at least once during the selection. We call this criterion *All_rules*. Note that this approach is based on two usual assumptions, namely the correctness of the formal specification and the uniformity hypothesis[1]. The uniformity hypothesis says that if a rule provides a correct answer for a single test case then it will provide correct answers for all the test cases that activate the rule. Of course, this assumption is strong and nothing can prevent it to be violated but recall that testing can only detect faults within an implementation and cannot prove the correctness of the implementation (as stated by Prof. E. Dijkstra).

Abstract test cases In the JSL formal model of the JCVM, a test case consists of a fully instantiated state of the VM and the valuation of several input parameters. However, it happens that several values of the state or several parameter values remain useless when testing a selected bytecode. To deal with these situations, the notion of abstract test case is used. In our case, an abstract test case represents a class of test cases that activate a given JSL function or equivalently a given CHR. The process which consists to instantiate an abstract test case to actually test an implementation is called concretization [2] and can be delayed until the test-execution time. For each CHR automatically generated, the goal is to find a minimal substitution of the variables (an abstract test case) that activate it. Covering a CHR consists in finding input values such as its guard would be satisfied. Hence, a constrained search process over the guards and the possible substitutions is performed. Before going to more details into this process, consider the CHRs of bytecode *push*. To activate *push_r1*, the states stack St must be empty whereas to activate *push_r2*, St must be rewritten into $cons\ H\ Lf$ (i.e. to posses at least one frame). Note that H, Lf, T and X are not constrained and do not require to be instantiated in the abstract test case. However, a randomised labelling can be used and to generate the two following concrete test cases, written under the form of JSL expressions[4]:
$(Bool, POS(XI(XO(XH))), Jcvm\_state(Nil, Nil, Nil))$ and
$(Byte, NEG(XH), Jcvm\_state(Nil, Nil, Cons(Frame(Nil, Nil, S(S(0)),$
$Package(0, S(0), Nil), True, S(0)), Nil)))$.

A constrained search process over the guards As usual in constraint programming, we would like to see the constraints playing an active role by exploiting the relations before labelling (test-and-generate approach). Note that this contrasts with classical functional testing techniques that usually instantiate first the variables and then check if they satisfy the requirements (generate-and-test approach).
Consider a CHR $\mathtt{r} : \mathtt{H} \Leftrightarrow \mathtt{G}|\mathtt{B}$ where $\mathtt{G} = p_1, \ldots, p_n$. Satisfying the guard $\mathtt{G}$ requires to satisfy at least one guard of the CHRs that define each predicate $\mathtt{p_i}$ of $\mathtt{G}$, i.e. finding a valuation such as $\mathtt{p_i}$ is simplified either in `true` or in a consistent conjunction of equalities. When $\mathtt{p_i}$ himself is a CHR call (deep guards), then its guard and body are also required to be consistent with the rest of the constraints. According to the *All_rules* testing criterion, the constraint store takes the following form:

$$\bigwedge_i \left(\bigvee_j (guard(\mathtt{p_i}, j) \wedge body(\mathtt{p_i}, j)) \right)$$

where $guard(\mathtt{p_i}, j)$ (resp. $body(\mathtt{p_i}, j)$) denotes the guard (resp. body) of the jth rule defining $\mathtt{p_i}$. Any solution of this constraint store can be interpreted as a test case that activates the CHR under test. Finding a solution to this constraint store leads to explore a possibly infinite search tree, as recursive or mutually recursive CHR are allowed. However, a simple occur-check test permits to avoid such problems. In this work, we followed a heuristic which consists to select first the guard with the easier guard to satisfy. A guard was considered easier to satisfy than another when it contains a smaller number of deep guards. The idea behind this heuristic is to avoid the complex case during the generation. This approach is debatable as these complex cases may contain the more subtle faults. See section 5.3 for a discussion on possible improvements. Note that the constraint store consistency is checked before going into a next branch, hence constraints allows pruning the search tree before making a choice. Note also that the test case generation process requires only to find a single solution and not all solutions, hence a breath-first search could be performed to avoid infinite derivations.

Oracles generation As the CHR specification of the JCVM is executable and the formal model is supposed to be correct, oracles can be generated just by interpreting the CHR program with generated test cases. For example, the following request gives us the oracle for the test

[4] $Jcvm\_state$, $Frame$, $Package$, XI, $X0$, XH, POS, $Byte$, NEG, $Bool$ and $True$ are JSL constructor symbols given in the JCVM formal model.

case generated for $push\_r1$:

```
?- push(bool,pOS(xI(xO(xH))),jcvm_state(nil,nil,nil)),R).
R=abnormal(jCVMError(eCode(state_error)),jcvm_state(nil,nil,nil))
```

Providentially, oracles can also be derived for abstract test cases. For example, oracle for abstract test case of $push\_r1$ is computed by the following request: `?- push(T,X,jcvm_state(Sh,He,nil)),R).`

```
R=abnormal(jCVMError(eCode(state_error)),jcvm_state(Sh,He,nil))
```

When delayed goals are present, a labelling process must be launched to avoid suspension. For example, the following request obtained by using the generated abstract test cases for $push\_r2$:

```
?- push(T,X,jcvm_state(Sh,He,cons(H,Lf))),R).
T=T, X=X, Sh=Sh, He=He, H=H, Lf=Lf, R=R
Delayed goals: push(T,X,jcvm_state(Sh,He,cons(H,Lf)),R)
```

requires R to be unified to $cons(\_X, \_S)$ to wake up the suspended goal. The labelling process can be based on deterministic or randomised[16] labelling strategies. In software testing approaches, random selection is usually preferred as it improves the flaws detection capacity. The simplest approach consists in generating terms based on a uniform distribution. Lot of works have been carried out to address the problem of uniform generation of terms and are related to the random generation of combinatorial structures [17]. In a previous work [18], we proposed a uniform random test cases generation technique based on combinatorial structures designs.

5.3 Experimental results

As previously said, the library `JSL2CHR.pl` generated 1537 CHRs that specify 45 JCVM bytecodes. The library generates a CHR program that is compiled by using the *ech* library of Eclipse Prolog [7]. We present the experimental results we obtained by generating abstract test cases for covering all the 443 CHRs associated to the bytecodes of the JCVM. These results were obtained on an Intel Pentium M at 2GHz with 1GB of RAM under Linux Redhat 2.6 The full process of generation of the abstract test cases for the 45 bytecodes (443 test cases) took 3.4s of CPU time and 47 Mbytes as the global stack size, 0.3 Mbytes as the local stack size and 2.6 Mbytes as the trail stack size. The detailed results for each bytecode are given in Tab.1, ordered by increasing number of abstract test cases (second column). Tab.1 contains the stack sizes as well as the CPU time (excluding time spent in garbage collection and system calls) required for the generation.

Analysis and discussion The approach ensures the coverage of each rule of the JSL bytecodes in a very short period of CPU time. The global and trail stacks remain stable whereas the local stack size increases with the number of test cases. A possible explanation is that some CHRs exit non-deterministically and allocation of variables cannot be undone in this case. We implemented a heuristic which consists to favour the CHRs that contain the smallest number of deep guards. This heuristic behaves well as shown by the short CPU time required for the bytecodes

Name	#tc	global stack (bytes)	local stack (bytes)	trail stack (bytes)	runtime (ms)
aload	1	33976048	148	1307064	0
arraylength	1	33849512	148	1299504	0
astore	1	33945864	148	1306660	0
invokestatic	1	33849512	148	1299504	0
nop	1	33945864	148	1306660	0
aconstnull	2	33854760	424	1300336	0
goto	2	33951112	424	1307492	0
jsr	2	33951112	424	1307492	0
push	2	33951112	424	1307492	0
conv	3	34055304	1076	1315924	10
dup	3	33972696	992	1310252	0
getfield	3	33876344	992	1303096	10
getfield_this	3	33876344	992	1303096	0
neg	3	34055304	1076	1315924	11
new	3	33971904	1020	1309932	11
pop	3	33972152	992	1310156	0
pop2	3	34074480	1076	1317828	0
putfield	3	33885840	1076	1303944	0
putfield_this	3	33876344	992	1303096	0
dup2	4	34122280	1884	1322772	10
swap	4	34029448	1884	1315948	11
ifnull	5	34023088	2216	1315392	10
ifnonnull	5	33926736	2216	1308236	10
icmp	6	34409440	3480	1343428	50
if_acmp_cond	6	34012528	3624	1316892	20
const	7	33968512	1512	1309716	0
invokespecial	7	34027448	4020	1317168	20
if_cond	8	34047496	3772	1319316	10
ret	8	34059064	3940	1320780	11
invokevirtual	9	34432272	7632	1349576	60
arith	11	33948080	836	1306660	0
athrow	11	34596760	7648	1364512	90
invokeinterface	11	35007240	12104	1394104	120
newarray	13	34073536	7604	1325612	20
return	13	34889544	11448	1386532	91
if_scmp_cond	14	34349752	11004	1351220	49
inc	18	34305392	9568	1344628	29
lookupswitch	18	34117768	9548	1332008	29
tableswitch	18	34117768	9548	1332008	31
load	19	34263232	11672	1343060	30
store	25	34752536	20108	1390876	81
checkcast	30	35053520	21384	1419380	280
getstatic	33	34408808	20652	1360596	60
putstatic	34	34944800	28660	1416196	120
instanceof	62	36468800	46964	1555588	580

Table 1. Memory and CPU runtime measures for each bytecode

that are specified with a lot of CHRs (`instanceof` is specified with 62 CHRs and only 0.6s of CPU time is required to generate the 62 abstract test cases). However, most of the time, this heuristic leads to generate test cases that put the JCVM into an abnormal state. In fact, in the JSL specification of the JCVM the abnormal states can often be reached by corrupting an input parameter. As a consequence, they are easy to reach. Although this heuristic is suitable to reach our test purpose (covering *All_rules*) and corresponds to some specific testing criterion such as *Test_all_corrupting_input* , it is debatable because it does not represent the general behaviour. Other approaches, which could lead to better test cases, need to be studied and evaluated. For example, selecting first the guard that contains the greatest number of deep guards could lead to build test cases that activate interesting parts of the specification. Finally, in these experiments, we only generated abstract test cases and did not evaluate the time required in the concretization step. Although, this step does not introduce research problems, considering it would allow to get a more accurate picture of test case and oracle generation with CHR. Thus, we could evaluate the efficiency of our approach and compare it to existing techniques.

6 Related Work

Bernot and al. [1] pioneered the use of Logic Programming to construct a test set from a formal specification. Starting from an algebraic specification, the test cases were selected using Horn clauses Logic. More recently, Gotlieb and al. [19] proposed to generate test sets for structural testing of C programs by using Constraint Logic Programming over finite domains. Given the source code of a program, a semantically-equivalent constraint logic program was built and questioned to find test data that cover a selected testing criterion. Legeard and al.[2] proposed a method for functional boundary testing from B and Z formal specifications based on set constraint solving techniques (CLP($\mathcal{S}$)). They applied the approach to the transaction mechanism of Java Card that was formally specified in B. Test cases were only derived to activate the boundary states of the specification of the transaction mechanism. Only Lötzbeyer and Pretschner [20, 21] proposed a software testing technique that uses CHR constraint solving. In this work, models are finite state automata describing the behaviour of the system under test and test cases are composed of sequence of input/output events. CHR is used to define new constraint solvers and permits to generate complex data types. Our work distinguishes by the systematic translation of formal specifications into CHRs. Our approach does not restrict the form of guards in CHR and appears so as more declarative to generate test cases.

7 Conclusion

In this paper, we have proposed to use the CHRs to generate functional test cases for a JCVM implementation. A JSL formal specification of the JCVM has been automatically translated into a CHR program and a test cases and oracles generation process has been proposed. The method permits to generate 443 test cases to test the 45 bytecodes formally specified. This result shows that the proposed approach scales up to a real-world example.
However, as discussed previously, other approaches need to be explored and evaluated. In particular, the coverage criterion *All_rules* initially selected appears as being too restrictive and other testing criteria could be advantageously used. Moreover, the test concretization step need to be studied in order to compare the efficiency of our approach against existing methods.
Finally, the key point of the approach resides in the use of deep guards, although their treatment needs to be evaluated both from the analytic and the experimental points of view.

8 Acknowledgements

We wish to acknowledge E. Coquery for fruitful discussions on CHR, and G. Dufay and G. Barthe who gave us the JSL formal model of the JCVM.

References

1. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications : a theory and a tool. Software Engineering Journal **6** (1991) 387–405
2. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. International Journal of Software Practice and Experience **34** (2004) 915–948
3. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (2002) 112–122
4. Barthe, G., Dufay, G., Huisman, M., Sousa, S.: Jakarta: a toolset for reasoning about JavaCard. In: Proceedings of E-smart 2001. Volume 2140 of LNCS., In I. Attali and T. Jensen Eds, Springer-Verlag (2001) 2–18
5. INRIA: The Coq proof assistant (1999) http://coq.inria.fr/.
6. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B., de Sousa, S.M.: A Formal Executable Semantics of the JavaCard Platform. In: Proceedings of ESOP'01. Volume 2028 of LNCS., D. Sands Eds, Springer-Verlag (2001) 302–319

7. Brisset, P., Sakkout, H., Fruhwirth, T., Gervet, C., Harvey, e.a.: ECLiPSe Constraint Library Manual. International Computers Limited and Imperial College London, UK. (2005) Release 5.8.
8. de Sousa, S.M.: Outils et techniques pour la vérification formelle de la plate-forme JavaCard. PhD thesis, Université de Nice (2003)
9. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Logic Programming **37** (1998) Special Issue on Constraint Logic Programming, In P. Stuckey and K. Marriott Eds.
10. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Cognitive Technologies. Springer Verlag (2003) ISBN 3-540-67623-6.
11. Duck, G., Stuckey, P., de la Banda, M.G., Holzbaur, C.: Extending arbitrary solvers with constraint handling rules. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP03). (79-90) 2003
12. Schulte, C.: Programming deep concurrent constraint combinators. In Pontelli, E., Costa, V.S., eds.: Second International Workshop on Practical Aspects of Declarative Languages. Volume 1753 of LNCS., Springer-Verlag (2000) 215–229
13. Podelski, A., Smolka, G.: Situated Simplification. In Montanari, U., ed.: Proceedings of the 1st Conference on Principles and Practice of Constraint Programming. Volume 976 of LNCS., Springer-Verlag (1995) 328–344
14. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B., de Sousa, S.M., Yu, S.W.: Formalization of the JavaCard Virtual Machine in Coq. In: Proceedings of FTfJP'00 (ECOOP Workshop on Formal Techniques for Java Programs), S. Drossopoulou and al, Eds (2000) 50–56
15. Dufay, G.: Vérification formelle de la plate-forme Java Card. PhD thesis, Université de Nice-Sophia Antipolis (2003)
16. Gouraud, S.D., Denise, A., Gaudel, M.C., Marre, B.: A New Way of Automating Statistical Testing Methods. In: Sixteenth IEEE International Conference on Automated Software Engineering (ASE). (2001) 5–12
17. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132** (1994) 1–35
18. Denise, A., Gaudel, M.C., Gouraud, S.D.: A Generic Method for Statistical Testing. In: Fifteenth IEEE International Symposium on Software Reliability Engineering. (2004) 25–34
19. Gotlieb, A., Botella, B., Rueher, M.: A CLP Framework for Computing Structural Test Data. In: Constraints Stream, First International Conference on Computational Logic. Number 1891 in LNAI, Springer-Verlag (2000) 399–413
20. Pretschner, A., Lötzbeyer, H.: Model Based Testing with Constraint Logic Programming: First Results and Challenges. In: Proceedings 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification. (2001)
21. Lötzbeyer, H., Pretschner, A.: AutoFocus on Constraint Logic Programming. In: Proceedings of (Constraint) Logic Programming and Software Engineering (LPSE'2000). (2000)

Le second article présenté ici se rattache au test symétrique, présenté dans le chapitre 4 de ce mémoire. Il est le fruit d'un travail collaboratif, mené dans le cadre du projet RNTL CASTLES, sur l'établissement d'une procédure de test croisé d'APIs pour Java Card. En effet, une des spécificités du travail relaté ici est l'exécution sur la Java Card des tests d'interface générés. Ceci a demandé la mise en oeuvre d'un kit–carte et l'écriture de générateurs de test directement exécutable sur la carte, ce qui est délicat à cause des limitations mémoire de la Java Card.

A. Gotlieb and P. Bernard. *A semi-empirical model of test quality in Symmetric Testing: Application to testing Java Card APIs*. **In Sixth International Conference on Quality Software (QSIC'06), Beijing, China, Oct. 2006.**

A Semi-empirical Model of Test Quality in Symmetric Testing: Application to Testing Java Card APIs *

Arnaud Gotlieb
IRISA-INRIA
Campus Beaulieu
35042 Rennes Cedex, FRANCE
Arnaud.Gotlieb@irisa.fr

Patrick Bernard
OBERTHUR CARD SYSTEMS
rue Auguste Blanche
92800 PUTEAUX, FRANCE
p.bernard@oberthurcs.com

ABSTRACT

In the smart card quality assurance field, Software Testing is the privileged way of increasing the confidence level in the implementation correctness. When testing Java Card application programming interfaces (APIs), the tester has to deal with the classical oracle problem, i.e. to find a way to evaluate the correctness of the computed output. In this paper, we report on an experience in testing methods of the Oberthur Card Systems Cosmo 32 RSA Java Card APIs by using the Symmetric Testing paradigm. This paradigm exploits user-defined symmetry properties of Java methods as test oracles. We propose an experimental environment that combines random testing and symmetry checking for (on-card) cross testing of several Java Card API methods. We develop a semi-empirical model (a model fed by experimental data) to help deciding when to stop testing and to assess test quality.

1. INTRODUCTION

Although formal verification and software testing were viewed as opposites for a long time, with formal verification concentrating on proving program correctness while testing concentrating on finding faults in program implementation, they can now be considered as complementary techniques [1]. In the smart card field, software testing is required by the Common Criteria evaluation scheme [2] to increase the confidence level of the certifying authority in the implementation correctness of security functions. In this context, techniques and tools that permit to automate (even partially) the testing process are welcome. Research works in that field include the BZ-testing approach designed by Legeard et al. [3, 4] to generate automatically test cases from a formal B or Z specification. The corresponding tools suite has been employed to validate the Java Card transaction mechanism by generating test cases on the boundary states of the formal specification [5]. In [6], Pretschner et al. followed a similar approach by using the AUTOFOCUS tool for specifying the command/response mechanism of an inhouse smart card and generating test cases for validating the authentification protocol of the card. At the same time, Clarke et al. [7] developed symbolic test generation algorithms and applied them to generate on-the-fly test cases for a feature of the CEPS[1] e-purse application and Martin and Du Bousquet [8] proposed to use UML-based tools to generate test suites for testing Java Card applets.

All these approaches have in common to require first a formal model (Z or B specification, automata, input/output transition system or statecharts) to be constructed in order to generate test cases. When the time-to-market of a new product is critical, this effort appears as being too costly and cheaper (but still rigorous) approaches are needed. Techniques such as statistical testing [9–11], boundary testing [12], or local exhaustive testing [13] do not require a formal model to be developed. Statistical testing aims at selecting randomly the values inside the input domain of the application under test by using pseudo-random numbers generators, boundary testing relies on selecting the boundaries of an input space partition, whereas local exhaustive testing systematically explores a bounded part of the input domain. In these approaches, testing just depends on the availability of oracles, that is, some procedures for predicting the expected results of the applications under test. Unfortunately, as earlier pointed out by Weyuker [14], there are programs to be tested for which the design of oracles is a non-trivial task. Examples of such programs in the smart card field include standard and proprietary Java Card APIs as they are just usually described by their interfaces and a few lines of natural text[2]. For these APIs, current industrial practices rely on coding the oracle as the result of another program that will be confronted with the result of the API under test. This approach suffers from several drawbacks such as the high cost of the development of oracles and the existence of faults into the oracles.

Recently, we have proposed [15] to address this oracle problem for Java programs by using user-defined symmetries of programs to check the correctness of the computed output. Here, symmetries are input-output permutation relations over program executions that lead to partitioning the input space into equivalence classes and the equivalence between two executions serves as an oracle. We introduce a testing paradigm called Symmetric Testing, where automatic test data generation was coupled with symmetries checking and local exhaustive testing to uncover faults inside the programs.

In this paper, we report on an experience in applying Symmetric Testing to test methods of the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API [33] by using random testing. Unlike our previous work [15], we develop here an original semi-empirical model to help decide when to stop testing and to assess test quality in Symmetric Testing. This model is fed with an empirical parameter (based on symmetry checking) in a theoretical model of random testing, in order to obtain the minimum number of test data required to reach a given level of quality. From the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API [33], we have selected the methods to test by studying their symmetry properties, as Symmetric Testing is only suitable for testing programs that possesses input-output symmetry relation. By using several tools, we have designed an experimental environment to build our semi-empirical model and to apply Symmetric Testing in situations as close as possible to the real situations. In contrast with other research works in testing Java card programs [6, 7], test execution and symmetries checking have been conducted by cross-testing on a smart card and not by using simulations.

The rest of the paper is organized as follows: section 2 presents the Symmetric Testing paradigm and gives examples of symmetry relations. Section 3 reports the symmetry analysis of a few methods of the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API while section 4 details our semi-empirical model of random testing based on symmetries checking. Section 5 reports the first experimental results and discusses extension of the framework to handle non-symmetric methods of the Java Card APIs. Finally section 6 pinpoints several perspectives to this work.

*Supported by the Réseau National des Technologies Logicielles as part of the CASTLES project (www-sop.inria.fr/everest/projects/castles/)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1]The Common Electronic Purse Specification is a standard for creating inter-operable multi-currency smart card e-purse systems.
[2]Although formalizations do exist [31].

2. SYMMETRIC TESTING

Exploiting symmetry in verification is not a new idea. Emerson and Sistla [17] and Ip and Dill [18] proposed early to exploit structural symmetries to address the problem of state explosion in model checking. This approach has been experienced and proved interesting in practice in several tools, such as VeriSoft [19] or SPIN [20]; its principle is based on basic results from group theory [17–19] and partial order techniques [21].

Based on similar ideas, we recently introduced Symmetric Testing [15] in the context of Software Testing. The flavour of our approach is explained here on a very basic example. Consider a program P intended to compute the greatest common divisor (gcd) of two non-negative integers u and v and suppose that P is tested with the following test datum $(u = 1309, v = 693)$ automatically generated by a random test data generator. Although we all know how to compute the gcd of two integers[3], it is not so easy to predict the expected value of $gcd(1309, 693)$ without the help of a calculator. Fortunately, gcd satisfies a simple symmetry relation: $\forall u \forall v, gcd(u, v) = gcd(v, u)$. So, if $P(1309, 693) \neq P(693, 1309)$ then the testing process will succeed to uncover a fault in P without the help of a complete oracle of gcd. Note that such a symmetry relation is a necessary but not sufficient condition, for the correctness of P. Such user-specified relations between several program executions have been called metamorphic relations and thoroughly investigated by Chen et al. [22–24].

Identifying such symmetry relations for larger programs might appear to be difficult or useless to detect non-trivial fault. On the contrary, we argue that numerous programs have to satisfy symmetry relations and these relations are useful for detecting subtle faults. In fact, every program P that takes an unordered set as argument has to satisfy a symmetry relation: the expected outcome of P is invariant under any permutation of the elements of the set. Numerous programs take unordered sets as arguments: consider sorting or selection programs that are used in search engines, programs that operate over data buffers, or graph-based programs just to name a few. Note that experimental evidence are also available to support this argument in [23, 24] and [15].

[3]With the Euclidian algorithm for example.

2.1 Symmetry relations

We generalized the above idea to obtain a formal and generic definition of symmetry relation. This definition is based on basic results from Group theory that are briefly recalled here. A detailed but still accessible presentation can be found in [25].

The notion of **symmetric group** is the corner-stone of Symmetric Testing. The symmetric group S_n is the set of bijective mappings from $\{1, .., n\}$ to itself. It has exactly $n!$ elements, called permutations. A permutation in S_n is written: $\theta = \begin{pmatrix} 1 & .. & n \\ i(1) & .. & i(n) \end{pmatrix}$ where $i(1), .., i(n)$ denote the images of $1, .., n$ by the permutation θ. A group action of S_n on a set E is a mapping $(\theta, x) \mapsto \theta \cdot x$ such as: $id_{S_n} \cdot x = x$ and $(\theta_1 \circ \theta_2) \cdot x = \theta_1 \cdot (\theta_2 \cdot x)$ for all $x \in E$ and $\theta, \theta_1, \theta_2 \in S_n$ (we say that S_n acts on E and θ acts on x). Note that E is closed under the action of S_n.

It is well-known that any permutation can be expressed as the composition of certain simple permutations, called cycles. Consider for example the permutation $\theta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$ of S_5, the same permutation can be written as $\theta = (1\ 3)(2\ 4\ 5)$ where each pair of brackets denotes a **cycle** $(a_1\ a_2\ ..\ a_r)$, that maps a_1 to a_2, .. a_{r-1} to a_r, a_r to a_1 and leave unchanged the other elements. A cycle written $(a_i\ a_j)$ is usually referred to as a **transposition**.

A subset X of elements of a symmetric group S_n is a set of generators iff every element of S_n can be written as a finite composition of the elements of X. For example, S_3 is generated by the two transpositions $\tau_1 = (1\ 2)$ and $\tau_2 = (2\ 3)$. More generally, S_n is generated by the transposition $\tau = (1\ 2)$ and the cycle $\sigma = (1\ 2\ ..\ n)$ and cannot be generated by less than two permutations [25]. Note that other two-generator sets can be found for S_n.

Symmetries of the function computed by a program P become interesting with regards to testing when they express general abstract properties. This leads to the notion of symmetry relations for a program.

DEFINITION 1 **(symmetry relation)**. *Let P be a program that computes a function f over an input domain $dom(f)$ toward a range domain $ran(f)$, and let S_n act on $dom(f)$ with a group action $\cdot$ and S_m act on $ran(f)$ with a group action $\odot$. A symmetry relation $\Psi_{n,m}$ holds for P iff*

1. *$\forall \theta \in S_n, \exists \eta \in S_m$ such that $\forall x \in dom(f)$, $f(\theta \cdot x) = \eta \odot f(x)$*

2. *$\Psi_{n,m} : \theta \longmapsto \eta$ is a homomorphism from S_n to S_m*

The first item requires f to satisfy an invariant property for all θ in S_n and for all x in the input domain of f. Note that η, the image of θ by the symmetry relation $\Psi_{n,m}$, is independent of the choice of x. Most of the time the two group actions will be the same ($\cdot \equiv \odot$), however we will see below an example of distinct group actions in a symmetry relation. The second item requires the symmetry relation to be a homomorphism. A homomorphism is a map φ from G_1 to G_2 such that $\varphi(\theta \circ \theta') = \varphi(\theta) \circ \varphi(\theta')$ for all $\theta, \theta' \in G_1$. Informally speaking, this requirement guarantees the symmetric structure of $dom(f)$ to be preserved by application of f, allowing so nice composition properties of symmetric relations. In our framework, we make an extensive use of this property to optimize the symmetry testing process, as explained below.

2.2 Examples

As an example, consider the Java Card program `void max3(byte[] A, byte[] B)` which selects the three maximum values of the array A and sorts them into the array B. If n denotes the size of A ($n \geq 3$) and f denotes the function computed by `max3` from $\mathbb{B}^n$ to $\mathbb{B}^3$, where $\mathbb{B}$ is the finite set of all possible bytes whose values are 8-bit signed two's complement integers, then the program `max3` has to satisfy a $\Psi_{n,3}$ symmetry relation because the array `B` is invariant under any permutation of `A`. Here, the considered group action (in both cases) is defined by: $S_n \times \mathbb{B}^n \rightarrow \mathbb{B}^n, (\theta, A = [a_1, .., a_n]) \mapsto \theta \cdot A = [a_{\theta^{-1}(1)}, .., a_{\theta^{-1}(n)}]$
As `B` is required to be sorted, all permutations θ will map to the identity of S_3.

As a more complicated example, consider the program `short getIndex0(short[] A)` that takes an array `A` of (non-negative) **distinct** values as argument and returns the index of the occurrence of 0 in the array or throws an exception if 0 is not present in `A`. The program `getIndex0` computes a function f from $\mathbb{S}^n$ to $\{0, 1, .., n-1\} \cup \{err\}$ where $\mathbb{S}$ denotes the finite set of 32-bit signed short integers, n denotes the size of `A` and err denotes an erroneous symbolic value. When 0 belongs to the array `A`, `getIndex0` has to satisfy a $\Psi_{n,n}$ symmetry relation because, 1) for any $\theta \in S_n$, $f(\theta \cdot A) = \theta \odot f(A)$ for all $A \in \mathbb{S}^n$ that contains an occurrence of 0, and 2) the identical map $\theta \longmapsto \theta$ is a group homomorphism. For instance, if
$A = [98, 4578, 1258, 654, 2589, 558, 12577, 0, 4876, 25541]$ and $\theta = (1\ 2\ ..\ 10)$ then $f(A)$ returns 7 and $f(\theta \cdot A)$ returns 8 which is the image of 7 by θ when it acts over $\{0, 1, .., 9\}$. Note that this example shows two distinct group actions: S_n acts over $\mathbb{S}^{10}$ when it is applied to the input sequence A of f whereas it acts over $\{0, 1, .., 9\}$ when applied to the outcome of f with the following group action:
$S_n \times \{0, 1, .., n-1\} \rightarrow \{0, 1, .., n-1\}$, $(\theta, x) \mapsto \theta \odot y = \theta(x)$.

2.3 Symmetric Testing

Symmetry relations can be used to seek for a subclass of faults within an implementation. Informally speaking, the Symmetric Testing principle aims at finding counter-examples (called symmetry violations) of the symmetry relation that a program has to satisfy.

DEFINITION 2. *(**Symmetry violation**) let P be a program over an input domain D and $\Psi_{k,l}$ be a symmetry relation that P has to satisfy, then a symmetry violation for P w.r.t. $\Psi_{k,l}$ is a couple (x, θ) such as $x \in D, \theta \in S_n$ and $P(\theta \cdot x) \neq \Psi_{k,l}(\theta) \odot P(x)$.*

The interesting point here is that symmetry violation can be checked by program executions whereas trying to prove formally that the function computed by a program satisfies a symmetry relation would be very difficult. Note that there is no way to distinguish among the two test data x and $\theta \cdot x$ the one that leads to an incorrect outcome for P. In the worst case, they can even be both faulty. So, given a set of test data and a symmetry relation, we get a naive procedure that can uncover a subclass of faults in P: it requires to compute P with all the permutations of the permutable inputs of each vector x in the test set and then to check whether the outcome vectors are equal to a permutation of the vector returned by P. The latter operation is called an *outcome comparison* in the rest of the paper.

However, the somehow naive procedure given above requires an outcome comparison for each possible permutation in the Symmetric Group S_n and, as S_n contains $n!$ permutations, the approach becomes impractical when n increases. The following result is exploited to reduce the number of outcome comparisons:

THEOREM 1. *Let P be a program that computes a function f and $\Psi_{k,l}$ be a symmetry relation for P, let $\tau = (1\ 2)$ and $\sigma = (1\ 2\ ..\ k)$, then we have*

$$\begin{cases} f \circ \tau = \Psi_{k,l}(\tau) \circ f \\ f \circ \sigma = \Psi_{k,l}(\sigma) \circ f \end{cases} \iff f \circ \theta = \Psi_{k,l}(\theta) \circ f \quad \forall \theta \in S_k$$

A proof of this theorem can be found in [15]; it is based on the fact that $\Psi_{k,l}$ is required to be a group homomorphism. Hence, by showing that $f(\tau \cdot x) = \Psi_{k,l}(\tau) \odot f(x)$ and $f(\sigma \cdot x) = \Psi_{k,l}(\sigma) \odot f(x)$, we get $f(\theta \cdot x) = \Psi_{k,l}(\theta) \odot f(x)$ for all $\theta \in S_k$, meaning that only two permutations are required to be checked. Moreover, by noticing that if (x, θ) is a symmetry violation then $(\theta \cdot x, \theta^{-1})$ is automatically another symmetry violation, the input domain to be explored can even be shrinked. These properties are exploited to design an efficient procedure for Symmetric Testing, that is fully described in [15].

The rest of the paper reports on our experience in applying Symmetric Testing combined with Random Testing to the testing of some Java Card API methods.

3. SYMMETRY IN JAVA CARD API

Unlike other smart cards, a Java Card includes a Java Virtual Machine and a set of API classes implemented in its read-only memory part. The Java Card Virtual Machine provides the interpretation of Java Card language constructs and the APIs are a set of classes and interfaces providing additional functionality that can be accessed by Java Card applets. A complete view of the development process of Java Card applets can be found in [29]. The OCS[4] Cosmo 32 RSA V3.4 (called Cosmo in the following) contains an implementation of the Java Card APIs.

3.1 The Cosmo Java Card APIs

The structure of the Cosmo Java Card platform is given in Fig.1. It consists of several components, such as an implementation of the Java Virtual Machine, the open platform applications, a set of packages implementing the standard SUN's Java Card API [16] and a set of proprietary packages. The four OCS proprietary packages consists of standard security services such as the VISA Open Platform Provider Security Domain, a set of base classes for implementing a Provider Security Domain, a complete range of classes for creating, maintaining and inspecting the card file-system and methods that are useful for JCRE related operations. Note that the Cosmo Java Card platform includes garbage collection facilities.

3.2 Symmetry analysis of a selected Cosmo Java Card API class

Among several possibilities, we selected `com.oberthurcs.javacard.file.Utilfs` as a case study because it consists of several generic utility methods that present symmetries. All the methods operate on byte or object arrays and are useful for dealing with APDU[5] buffers. The `Utilfs` class is composed of 10 methods, shown in Tab.1 together with their symmetry relations. The first and second column are extracted from the Cosmo API informal specification [33]. The third column summarizes the results of our symmetry analysis. The set of all possible bytes is noted $\mathbb{B}$ and the set of available objects is noted $\mathbb{O}$. Note first that some methods that deal with multiple array elements are tagged as NonAtomic. Atomicity defines how the card handles the contents of persistent storage

[4]Stands for Oberthur Card Systems

[5]Application Protocol Data Unit is an ISO-normalized communication format between the card and the off-card applications.

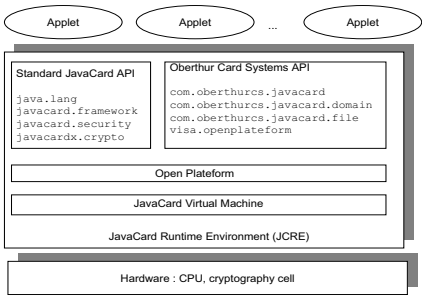


Figure 1: The OCS Cosmo 32 RSA V3.4 platform

after a failure or fatal exception during an update of a class field or an array component [16]. An applet might not require atomicity for array updates. The `Utilfs.arrayAndNonAtomic` method is an example: it shall not use the transaction commit buffer even when called with a transaction in progress.

Among the ten methods of this class, we found that seven have to satisfy a simple symmetry relation. We discuss a few of them; the other ones can easily be deduced from these. The method `short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)` can be abstracted by a function f from $\mathbb{B}^{len} \times \mathbb{B}^{len}$ to $\mathbb{B}^{len}$ as it modifies the input-output parameter `dest` by combining `src` and `dest` and by considering all other parameters as non-variable. `arrayAndNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation because of the following invariant property:

$$\forall \theta \in S_{len}, f(\theta \cdot dest, \theta \cdot src) = \theta \odot f(dest, src)$$

This is due to the fact that $dest$ and src represent two unordered sets of values for this method. Note that the two group actions are distinct as the first one holds over the set $\mathbb{B}^{len} \times \mathbb{B}^{len}$ (i.e. $\theta \cdot (x, y) = (\theta \cdot x, \theta \cdot y)$ where x and y are vectors of $\mathbb{B}^{len}$) whereas the second one $\odot$ holds just over $\mathbb{B}^{len}$. The methods `short arrayCompare( byte[] src, short srcOff, byte patByte, short length)` and `short arrayFindByte(byte[] src, short srcOff, short len, byte pattern)` have each to satisfy a $\Psi_{len,len}$ symmetry relation as the following invariant property holds: $\forall \theta \in S_{len}, f(\theta \cdot src) = \theta \odot f(src)$ where f denotes a map from $\mathbb{B}^{len}$ to $\{1, .., len\}$. In fact, these two programs are selection programs that are invariant to permutations of a subset of their input parameters. In the `Utilfs` class, some methods do not have to satisfy simple symmetry relations. For example, the method `arrayFindShort` has incompatible input types, that is to say the method looks for a short integer variable into an array of bytes. Although we have not realized a full study of the Cosmo Java Card APIs, we took a look at other classes to find symmetry relations. For example, the classes `visa.openplatform.OPSystem`, `javacard.security.MessageDigest` or `javacard.framework.Util` contain methods that have to satisfy symmetry relations. Note that the compositional definition of symmetry relations allows to combine several method calls. However, we also found numerous classes where the symmetry analysis does not reveal any symmetry relations. Examples of such classes include `com.oberthurcs.javacard.file.*` or `javacard.framework.JCSystem`. So, the Symmetric Testing approach remains limited in application to a restricted part of the Cosmo Java Card APIs. For these classes and methods, other input-output properties should be taken as partial oracles as discussed in section 5.3.

4. A SEMI-EMPIRICAL MODEL

In this paper, the Symmetric Testing principle combines random test data generation and automatic symmetry checking. Random testing has traditionally been viewed as a blind approach of program testing. However, results of actual random testing experiments confirmed its potential to reveal faults and as a validation tool [9]. Nevertheless, when the tester wants to exploit a random test data generator, he faces two main difficulties. The first is the classical oracle problem already discussed in the introduction of this paper: an automatic way of checking the output correctness is required. The second problem is to determine the test quality level reached by such a testing approach. In general, it is difficult to quantify how reliable is a program that has only been tested by randomly generated test data. Several works deal with this problem by using a purely theoretical framework based on probabilistic analysis [10, 30]. In this paper, we exploit a semi-empirical model (a model fed by experimental data) to help decide when to stop testing. This section is devoted to the presentation of this semi-empirical model.

4.1 Random testing

Let p_f be the probability that a randomly generated input test datum x exhibit a fault in the program P. A fault in P can be understood as a syntactical change in the source code that leads, for some input data, to a difference between $P(x)$ and the expected output of the function computed by P with x. By a simple probabilistic reasoning, a model of random testing based on p_f can be developed. It is a law between the number N of randomly generated test data and a probabilistic parameter that characterizes the fault-detecting effectiveness of the random testing strategy [9, 32]. The probability of detecting at least one failure is called the test quality[6] and it is noted Q_N [10]. Its value is given by the following definition:

DEFINITION 3. $Q_N = 1 - (1 - p_f)^N$

As an immediate consequence, we get an estimation of the minimum number of test data required to reach a certain value of Q_N:

THEOREM 2. $N \geq \lceil \frac{ln(1-Q_N)}{ln(1-p_f)} \rceil$

where $\lceil x \rceil$ denotes the ceiling function applied to a real number x.

4.2 The empirical parameter p_s

The above model of random testing suffers from a major drawback: it is based on p_f which is almost impossible to evaluate without a precise knowledge of all the existing faults in the program P. We address this problem by using an empirical parameter in place of p_f to build our model. This parameter p_s is related to symmetry checking: p_s is the probability of detecting a symmetry violation (x, θ) when x is randomly generated over a subset of size s of the input domain. This parameter characterizes the probability for Symmetric Testing to reveal a fault in P when it makes use of a random test data generator to generate a single test datum.

In this paper, we propose to empirically evaluate p_s on a correct specimen program by making use of fault-injection techniques.

[6]This measure is also called the P-measure

Table 1: Symmetry in the OCS Utilfs methods

Java Card methods	Informal specifications	Symmetry relations
`short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise AND on the first operand dest and the second operand src into dest	The program computes the following function: $f : \mathbb{E}^{len} \times \mathbb{E}^{len} \longrightarrow \mathbb{E}^{len}$ $(dest, src) \longmapsto dest'$ where $dest'$ stands for the value $dest$ computed after application of the `arrayAndNonAtomic` program. It has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayCompare( byte[] src, short srcOff, byte patByte, short len)`	Returns the index of the first byte in the specified part of src that does **not** match patByte, or 0xFFFF if every byte matches	We ask src to contain 1 occurrence of patByte and $srcOff = 0$ $f : \mathbb{E}^{len} \longrightarrow \{1, .., len\}$ $src \longmapsto ret$ `arrayCompare` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayFindByte( byte[] src, short srcOff, short len, byte pattern)`	Returns the index of the first byte in the part of src that matches the specified pattern.	We ask src to contain 1 occurrence of pattern $f : \mathbb{E}^{len} \longrightarrow \{1, .., len\}$ $src \longmapsto ret$ `arrayFindByte` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayFindPattern( byte[] src, short srcOff, short srcLen, byte[] patSrc, short patOff, short patLen)`	Returns the index of the first byte in the part of src that matches the specified pattern.	no simple symmetry
`short arrayFindShort( byte[] src, short srcOff, short len, short pattern)`	Returns the index of the first byte in the part of src that matches the specified pattern (a short is 2 bytes).	no simple symmetry
`short arrayOrNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise OR on the first and second operands into dest.	$f : \mathbb{E}^{len} \times \mathbb{E}^{len} \longrightarrow \mathbb{E}^{len}$ $(dest, src) \longmapsto dest'$ `arrayOrNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayXorNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise XOR on the first and second operands into dest.	$f : \mathbb{E}^{len} \times \mathbb{E}^{len} \longrightarrow \mathbb{E}^{len}$ $(dest, src) \longmapsto dest'$ `arrayXorNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation
`short getObjectIndex( java.lang.Object[] src, short srcOff, short n, java.lang.Object pattern)`	Returns the index of the nth occurrence in the part of src that matches pattern.	len denotes the length of src and $n = 1$, we ask src to contain 1 occurrence of pattern $f : \mathbb{O}^{len} \longrightarrow \{1, ..., len\}$ $src \longmapsto ret$ `getObjectIndex` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short getShortIndex( short[] src, short srcOff, short n, short pattern)`	Returns the index of the nth occurrence in the part of src that matches pattern.	len denotes the length of src and $n = 1$, we ask src to contain 1 occurrence of pattern $f : \mathbb{O}^{len} \longrightarrow \{1, ..., len\}$ $src \longmapsto ret$ `getShortIndex` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short getTaggedShort( byte[] src, short srcOff, short srcLen, short tag)`	Returns the index of the first TLV tag in the part of src that matches the specified tag.	no simple symmetry

Using a specimen program is advantageous as we can easily inject faults by modifying its source code. The key point of our approach resides in the knowledge of symmetry violations occurring when checking the output correctness of this program. Note that this approach is based on an uniform hypothesis: the inferred value for the specimen program applies to other programs as well. This hypothesis is debatable and relates to the difficulty of finding a representative sample in statistics. In our framework, we preferred to select a single representative program rather than a large set of non-representative programs. Of course, any other more representative program can be employed. For example, when testing an airborne flight-guidance software, one can employ a well-established correct program of the airborne software domain.

4.3 Our protocol to evaluate p_s

Our protocol to evaluate p_s is based on a set of faulty versions of the specimen program P that are automatically created by a mutation analysis scheme [34]. A mutant Q is a version of P where a single syntactical change has been introduced. Classically, a mutant is said to be killed by a test datum x when $Q(x) \neq P(x)$. In our framework, we will consider a mutant to be killed if there exists $\theta \in S_n$ such as (x, θ) is a symmetry violation for Q w.r.t. $\Psi_{n,m}$. The value of p_s depends on s, the size of the subdomain of this input space that is considered for the random test data generation. Given a size s, the empirical protocol is as follows:

1. built $\{Q_i\}_{i=1..K}$ a set of K mutants of a specimen program P that has to satisfy a symmetry relation $\Psi_{n,m}$;
2. for each Q_i, compute the booleans $b_{i,x} = (Q_i(\tau.x) \neq \Psi_{n,m}(\tau).Q_i(x)$ or $Q_i(\sigma.x) \neq \Psi_{n,m}(\sigma).Q_i(x))$ for each test datum x of the input domain of P;
3. returns $p_s = \frac{1}{N*K}.\sum b_{i,x}$ which is just the median value of the probability for the K mutants.

Note that the programs Q_i are executed on a large part of their input domain, hence it is important to select a specimen program having an input space of reasonable size. Note also that only two permutations are required to be checked in this protocol ($\tau = (1\ 2)$ and $\sigma = (1\ 2\ ..\ n)$). This is a direct consequence of Theorem 1.

We selected the well–known triangle classification program `trityp` [27], that belongs to the Software Testing folklore. It takes three non-negative bytes as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene, isocele, equilateral or illegal. The results of `trityp` must be invariant to every permutation of its three input values, leading to a $\Psi_{3,1}$ symmetry relation. This program appears to be an interesting specimen candidate as it contains a lot of decisions and the probability of a symmetry violation to occur is highly related to the flow of control. Hence, this probability highly depends on the input subdomain that is being explored. This property has recently been investigated from an experimental point of view in [24]. Of course, any more representative program can be employed but we would like just to study the feasibility of the approach rather than designing a fully acceptance testing methodology.

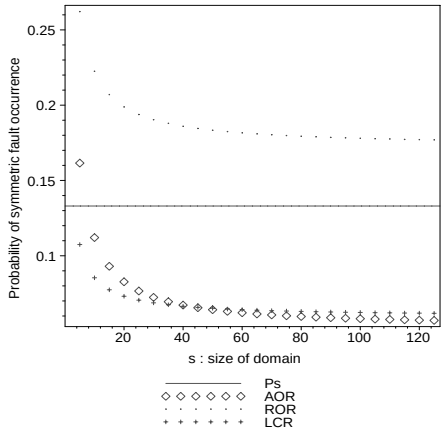


Figure 2: Empirical evaluation of p_s

In our empirical protocol, application of the tool MuJava [34] led to build automatically 36 mutants where an arithmetic operator was replaced (AOR), 85 mutants where a relational operator was replaced (ROR), and 14 mutants where a Logical connector was replaced (LCR). In the current MuJava framework, equivalents mutants[7] are not removed from the set of mutants, although they cannot be revealed by the means of testing [27]. So, 135 mutants of the `trityp` programs were built by the tool and the input domain was restricted to contain at most $s = 126^3 = 2000376$ input values. Among the 135 mutants, 21 were not killed by Symmetric Testing but we kept them in the experiments to avoid introducting a bias in the study.

For each mutant, we compute the number of symmetry violations found when exploring exhaustively a subdomain of the input domain. The average number of symmetry violations that were detected when exploring a subdomain of size s allows for calculating the probability of a symmetry violation to occur by using a uniform random test data generator (p_s). Fig.2 contains the results we got for several increasing values of s ($1^3, 6^3, 11^3, ..., 126^3$) by distinguishing the class of considered mutants (AOR,ROR,LCR). We compute p_s as the center of mass of the 3 bottom values obtained for the greatest size s ($s = 126^3$). Hence, $p_s = (36 * 0.057 + 85 * 0.177 + 14 * 0.062)/135 = 0.133$.

4.4 Test quality based on symmetry violations

Based on definition 3, we get that $q_N = 1-(1-p_s)^N$ for random testing based on symmetry checking. The test quality q_N differs from Q_N as q_N is only based on symmetry checking. In fact, q_N measures the probability of N randomly generated test data in a subdomain of size s to reveal at least one symmetry violation in P. When the property is enforced (P has been tested with a test quality q_N), we get that the symmetry relation is satisfied by the program P with a probability q_N. So, by using this model it becomes possible to assess the symmetry-based test quality for P.

[7] programs which compute the same outcome as the original program although a mutation operator is applied

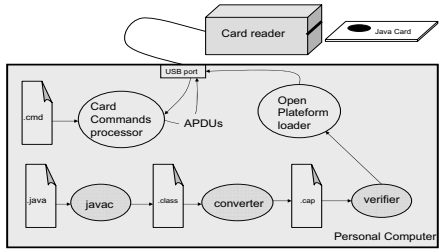


Figure 3: Experimental environment

The test quality was required to be equal to 0.9995 as is usually the case in experimental frameworks [10]. By using the empirical value of $p_s = 0.133$ and the theorem $N \geq \lceil \frac{ln(1-q_N)}{ln(1-p_s)} \rceil$, we get that $N = 54$, meanning that at least 54 test cases must be generated. Note that we have just argued that this (arbitrary) value is suitable for feeding our semi-empirical model.

5. EXPERIMENTAL ENVIRONMENT

The goal of the experiments was to study the applicability of Symmetric Testing to reveal faults within Java Card APIs. The validation process of Java Card APIs is usually made of three distinct phases: firstly, Java card test applets are developed on a host machine by using simulation libraries; secondly, the tests are applied to an emulation code that runs on a card emulator; and finally, the test execution is conducted by cross-testing on the Java card. Our experiments were performed in situations as close as possible to the real usage. Hence, test execution and symmetries checking have been conducted by cross-testing on the Java card with the help of a card reader. In this respect, we differ from other smart cards testing research approaches that focus only on test cases generation [6, 7]. In fact, we would like to check whether Symmetric Testing can be combined with Random Testing in a cross-testing environment, which was a challenging question as lots of limitations to memory resources arise in such situations. Moreover, this approach required to develop carrefully our prototype implementation to masterize the memory and time consumption. Fig.3 contains a view of our experimental environment. It is composed of five components: the java compiler (SUN SDK 1.4), the OCS converter that produces standard Java Card byte code (converted applet file), the OCS verifier which statically determines whether a cap file complies with the Java Card specifications, the Open Platform loader which downloads and manages the applets onto the card and a Card Command Processor that sends commands to the smart card via a card-reader interface. Note that the bytecode verification process is done off-card by the OCS verifier. The Card Command Processor is a command interpreter that accepts several language constructs such as conditional and loop.

5.1 Tests generation and execution

In our experimental environment, a special attention has been paid to minimize the communications between the reader and the card. Recalling that our goal was to realize the test execution and the symmetry checking processes on-card, passing large sets of random numbers through the APDU mechanism would have been too

time-consuming. Hence, we have designed a Java Card applet (to be loaded on-card) that generates test data and that checks the symmetry relations. This applet serves as a test harness and its size is around 1.2 kbytes. It defines a single command `TEST_API` that launches three executions of the API method under test ($P(x)$, $P(\tau \cdot x)$, $P(\sigma \cdot x)$) and checks the computed output with regards to a given symmetry relation. The applet makes use of a uniform random test data generator provided by the Cosmo API implementation of the `javacard.security.RandomData` class to generate a single test datum x. In case of symmetry violation, a boolean is returned through the APDU mechanism to inform the tester. After having compiled, converted and verified the applet, it is loaded on-card by the OP loader. Then, the command `TEST_API` is launched $N = 54$ times with the help of a command script, interpreted by the Card Command processor.

5.2 Experimental results

For our experiments, we selected the seven methods from the Cosmo Java Card API `Utilfs` that have a symmetry relation to satisfy. In the industrial validation process of the Cosmo kit, these methods are systematically tested by using a few values. For instance, the `arrayAndNonAtomic` method is tested with two randomly generated byte arrays by varying the values of destOff, srcOff and len. By using the approach presented in the paper, we tested each method of the API `Utilfs` (that has to satisfy a symmetry relation) with 54 randomly generated test data[8]. Tab. 2 contains the time elapsed to pass all the 54 tests for each method. This time value corresponds to the absolute user time (including garbage collections, operating system calls, etc.) elapsed on the 8-bit CPU Cosmo processor. It is just given here to illustrate the interest of using symmetry relations as automatic (partial) test oracles. This time should be compared to the time required by the tester to predict the expected results of the methods with each of the 54 randomly generated test data.

The test quality achieved by these tests is equals to 0.9995, that is to say each of these methods satisfies its symmetry relation with a test quality of 0.9995. We did not find any symmetry violations during this testing process but this does not prove the absence of symmetry violations as our approach is only probabilistic.

5.3 Discussion and further work

The main limitation of the Symmetric Testing paradigm arises when one tries to apply it to non-symmetric methods [15]. To address this problem, we plan to explore other properties to check the output correctness of Java Card APIs. Recently, Chen et al. proposed in [22] to use existing relations over the input data and the computed outcomes to eliminate faulty programs. Formally speaking, let $\{I_1, .., I_n\}_{n>1}$ be n distinct test data for a program intended to compute a function f and suppose that given a relation r over $\{I_1, .., I_n\}$, the results $f(I_1), .., f(I_n)$ must satisfy a property r_f, then we have: $r(I_1, .., I_n) \implies r_f(f(I_1), .., f(I_n))$. These relations, called metamorphic relations, are more general than symmetry relations. In a previous work we did [28], we proposed to automate the generation of input data that violate a given metamorphic relation, by using Constraint Logic Programming techniques. Specifying such metamorphic relations over the Java Card APIs would be interesting as they could serve as (partial) test oracles for non-symmetric methods. A similar approach would be to consider formally specified postconditions as a way to check the output correctness. For example, the formal specification of Java Card APIs written in JML (Java Modeling Language) by Poll et al. [31] could

[8]All the randomly generated arrays are of size `0x7F` which is the greatest byte value

```
ensures (\forall int i; (i<=0 & i<dest.length)
==>  (destOff <=i & i<destOff+length) ?
      dest[i] == src[srcOff + (i-destOff)] :
      dest[i] == \old(dest[i]) );
ensures \result == destOff+length ;
```

Figure 4: JML postconditions for `arrayCopy`

be an interesting way of getting formulas that can serve as (partial) test oracle. However, combining these formal postconditions with a random test data generator remain a non-trivial task as they make use of specific constructs that limit the possibility to asses test quality. For example, the formal JML postcondition of the `arrayCopy` Java Card API method extracted from [26] and shown in Fig.4 makes use of array accesses and a loop construct for which a fault occurrence probability seems to be difficult to establish.

6. CONCLUSION

In this paper, we have introduced a software testing framework for on-card testing of symmetric Java Card API methods. The framework contains a semi-empirical model to help deciding when to stop testing and how to assess test quality. We have reported on a first experience on testing a few methods of the OCS Cosmo 32 RSA V3.4 Java Card API by using the Symmetric Testing paradigm. Further work will be dedicated to the exploitation of non-symmetric properties to check the output correctness of Java Card methods, such as metamorphic relations or postconditions extracted from a formal specification. Another perspective will consist in exploring how Symmetric Testing can be tuned to deal with the resources consumption problem. Due to its limited memory and execution features, Java cards and Java Card APIs must be thoroughly tested w.r.t. memory and time consumption. Symmetry relations combined with random testing could be an interesting candidate to find counter-examples of statically estimated consumption bounds but this remains to be shown.

7. REFERENCES

[1] J. P. Bowen, K. Bogdanov, J. Clark, M. Harman, R. Hierons, and P. Krause. FORTEST: Formal methods and testing. In *COMPSAC 02: 26th IEEE Annual Int. Computer Software and Applications Conf., Oxford, UK*, pages 91–101. Computer Society Press, Aug. 2002.

[2] ISO International Standard 15408. *Common Criteria for Information Technology Security Evaluation*, Aug. 1999. CCIMB-99-033, Part 3: Security assur. req.

[3] B. Legeard and F. Peureux. Generation of functional test sequences from b formal specification : presentation and industrial case study. In *In Proc. of ASE'01*, IEEE Computer Society Press, pages 377–381, San Diego, USA, Nov. 2001.

[4] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. Bz-testing-tools: A tool-set for test generation from z and b using constraint logic programming. In *In Proc. of FATES'02, Formal App. to Testing of Software, Workshop of CONCUR'02*, pages 105–120, Brn, Czech Republic, Aug. 2002.

[5] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In *In Proc. of FME'02, Formal Methods Europe*, Springer Verlag LNCS 2391, pages 21–40, Copenhaguen, Denmark, Jul. 2002.

[6] A. Pretschner, O. Slotosch, H. Ltzbeyer, E. Aiglstorfer, and S. Kriebel. Model based testing for real: The inhouse card case study. In *In Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, pages 79–94, Paris, Jul. 2001.

Table 2: Symmetry in the OCS Utilfs methods

Java Card methods	User time elapsed
`short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	13 min 59 sec
`short arrayCompare( byte[] src, short srcOff, byte patByte, short length)`	6 min 15 sec
`short arrayFindByte( byte[] src, short srcOff, short len, byte pattern)`	4 min 24 sec
`short arrayOrNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short length)`	13 min 57 sec
`short arrayXorNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short length)`	13 min 55 sec
`short getObjectIndex( java.lang.Object[] src, short srcOff, short n, java.lang.Object pattern)`	2 min 01 sec
`short getShortIndex( short[] src, short srcOff, short n, short pattern)`	2 min 05 sec

[7] D. Clarke, T. Jeron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *In Int. Conf. on Research in Smart Cards (e-Smart'01)*, Springer Verlag, LNCS 2140, pages 58–70, 2001.

[8] H. Martin and L.d. Bousquet. Automatic test generation for java-card applets. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security, First International Workshop, JavaCard 2000, Cannes, France, September 14, 2000, Revised Papers*, volume 2041 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2001.

[9] J.W. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Trans. on Soft. Eng.*, 10(4):438–444, Jul. 1984.

[10] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Journal of Sotware Testing, Verification and Reliability*, 1(2):5–25, July 1991.

[11] T. Y. Chen, T. H. Tse, and Y. T. Yu. Proportional sampling strategy: a compendium and some insights. *The Journal of Systems and Software*, 58 (2001), pages 65–81. Elsevier 2001.

[12] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company, INC., Menlo Park, CA, 1987.

[13] T. Wood, K. Miller, and R. E. Noonan. Local exhaustive testing: a software reliability tool. In *Proc. of the Southeast regional conf.*, pages 77–84. ACM Press, 1992.

[14] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4), 1982.

[15] A. Gotlieb. Exploiting symmetries to test programs. In *IEEE International Symposium on Software Reliability and Enginering (ISSRE)*, pages 365–374, Denver, CO, USA, Nov. 2003.

[16] SUN Microsystems. *Java Card 2.1.1 Application Programming Interface*, May 2000.

[17] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[18] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design: An International Journal*, 9(1/2):41–75, August 1996.

[19] P. Godefroid. Exploiting symmetry when model-checking software. *FORTE'99*, pp 257-275.

[20] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *SPIN*, pp 1–19, 2000.

[21] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *Soft. Tools for Tech. Transfer*, 2, 1998.

[22] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Fault-based testing in the absence of an oracle. In *IEEE Int. Comp. Soft. and App. Conf. (COMPSAC)*, pages 172–178, 2001.

[23] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[24] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *4th Ibero-American Symp. on Software Engineering and Knowledge Engineering (JIISIC 04)*, pages 569–583, 2004.

[25] M. A. Armstrong. *Groups and Symmetry, UTM*. Springer Verlag, 2nd ed., 1988.

[26] C.-B. Breunesse, N. Catao, M. Huisman, and B.P.F. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55(1-3):53–80, 2005.

[27] R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Trans. on Soft. Eng.*, 17(9):900–910, Sep. 1991.

[28] A. Gotlieb and B. Botella. Automated metamorphic testing. In *27th IEEE COMPSAC'03*, Dallas, TX, USA, November 2003.

[29] U. Hansmann, M.S. Nicklous, T. Schack, and F. Seliger. *Smart Card Application Development using Java*. Springer Verlag, 2000.

[30] Y. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich. Software reliability growth with test coverage. *Trans. on Reliability*, 51(4):420–426, Dec. 2002.

[31] H. Meijer and E. Poll. Towards a full formal specification of the java card api. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001*, LNCS 2140, pp 165–178. Springer Verlag, 2001.

[32] S. Ntafos. On Random and Partition Testing. *Soft. Eng. Notes*, 23(2):42–48, 1998.

[33] Oberthur Card Systems. *Cosmo 32 RSA V3.4 API Reference Guide*, 2003.

[34] J. Offutt, Y. Ma, and Y. Kwon. An experimental mutation system for java. *Softw. Eng. Notes*, 29(5):1–4, 2004.

Ce travail, écrit en collaboration avec le responsable de la validation fonctionnelle des cartes produites chez Oberthur, a démontré la faisabilité du test symétrique sur la Java Card et son potentiel pour évaluer la correction de certaines APIs. Au delà du test symétrique, c'est le test exhaustif borné[1] qui est utilisé ici et il est intéressant de constater que cette forme de test a reçu un grand intérêt par la suite dans le domaine du test logiciel [Coppit 05].

[1] "exhaustive bounded testing"

Chapter 9

Système d'alerte et anti-collision (TCAS)

Les logiciels critiques de l'avionique civile ou militaire auxquels le chercheur peut avoir accès, sont très peu nombreux. En effet, le code source fait partie du secret industriel des grandes entreprises qui sont chargés du développement et de la maintenance de ces logiciels, et il n'est donc généralement pas accessible. Dans le cadre du projet RNTL CAT (*C Analysis Toolbox*, 2005-2009) qui avait pour objet la réalisation d'outils pour l'analyse et la vérification de programmes C, nous avons pu avoir accès à de tels logiciels. Dans ce contexte, une campagne d'expérimentations a été menée visant certains logiciels embarqués de l'avionique civile, certains étant fournis par nos partenaires industriels et d'autres provenant de source publique comme le SIR[1] [Do 05]. L'intérêt de ces derniers étant que les résultats obtenus soient publiables. Le fruit d'une de ces expériences, menées dans le cadre du projet CAT, est décrit dans l'article de ce chapitre.

A. Gotlieb. *TCAS software verification using constraint programming*. **The Knowledge Engineering Review, 2009. Accepted for publication.**

[1] *Software Infrastructure Repository*

The Knowledge Engineering Review, Vol. 00:0, 1–15. © 2009, Cambridge University Press
DOI: 10.1017/S000000000000000 Printed in the United Kingdom

TCAS software verification using Constraint Programming

Arnaud Gotlieb

INRIA - Rennes - Bretagne Atlantique
35042 Rennes Cedex, France
E-mail: Arnaud.Gotlieb@irisa.fr

Abstract

Safety-critical software must be thoroughly verified before being exploited in commercial applications. In particular, any TCAS (Traffic Alert and Collision Avoidance System) implementation must be verified against safety properties extracted from the anti-collision theory that regulates the controlled airspace. This verification step is currently realized with manual code reviews and testing. In our work, we explore the capabilities of Constraint Programming for automated software verification and testing. We built a dedicated constraint solving procedure that combines constraint propagation with linear programming to solve conditional disjunctive constraint systems over bounded integers extracted from computer programs and safety properties. An experience we made on verifying a publicly available TCAS component implementation against a set of safety-critical properties showed that this approach is viable and efficient.

1 Introduction

In critical systems, software is often considered as the weakest link of the chain and there are many stories of software bugs that yield catastrophic consequences. For instance, on 2004, September 14, L. Geppert reports[1] that the contact voice lost of Los Angeles air traffic controllers was due to a software bug that resulted to an unexpected shutdown of the Voice Switching and Control System. It turned out that shutdown was due to a 32-bit integer value running out of digits. Hopefully, in a situation that could have proved deadly, tragedy was avoided not only by both pilots and controllers professionalism, but also by another software-based critical system: the Traffic Alert and Collision Avoidance System (TCAS), which is embedded on aircraft to prevent from midair collisions. Thus, small software bugs can result to catastrophic situations and therefore, they must be tracked and eliminated from critical systems.

In the Avionics domain, software verification currently includes manual code review and analysis, unit testing, software and hardware integration testing and validation testing. These various tasks address distinct not-redundant verification levels and offer reasonable confidence in terms of correction, reliability and performance. Code reviews consist in reading and discussing code written by other developers and help checking assertions derived from safety-critical properties. Unit testing consists in executing pieces of code (units) in isolation of the rest of the system with the intent of finding bugs. Test cases are selected from the input domain and used to exercise the software unit ; then, a test verdict (either pass or fail) is produced by an oracle procedure to check the computed results. For testing software units in isolation, stubs have to be created to replace actual function calls. Software integration testing aims at testing the interaction between units and it is performed just by removing the stubs. Hardware integration testing consists in executing the code in its operational and physical environment. For avionics software, it means executing cross-compiled code on hardware emulators or embedded targets. Finally, validation testing aims at verifying high-level requirements through simulations and operational scenarios. As a

[1] IEEE Spectrum (`www.spectrum.ieee.org/nov04/4015`)

consequence of the increasing complexity of avionics software, it is usually acknowledged that these techniques can hardly scale-up (Randimbivololona 2001). Code reviews are human-based approaches that highly depend on the confidence of reviewers. Testing is faced with combinatorial explosion and manual test data generation rapidly becomes too hard. In addition, test data selection and oracle production are error-prone processes that require competent and experienced engineers.

In our work, we explore the capabilities of Constraint Programming for automating parts of the software verification process. For several years, we have concentrated our efforts on automating the test case generation process by using dedicated constraint solving procedures. Initially, we proposed using classical constraint propagation with bound-consistency filtering and labeling for handling integer computations (Gotlieb *et al.* 2000) and floating-point computations (Botella *et al.* 2006). Then, we refined the solving procedure with linear relaxations (Denmat *et al.* ISSRE 2007) and dedicated global constraint approaches (Denmat *et al.* CP 2007). We also addressed the extension of our approach to pointer variables (Gotlieb *et al.* 2007) and dynamic allocated structures. Recently, we have started to explore software verification with this approach and this paper mainly reports on our first experience in this matter.

In this paper, we present a general constraint solving procedure that combines bound-consistency filtering with linear programming over the rationals for solving disjunctive constraint systems over bounded integers. These constraint systems are extracted from imperative programs and correspond to various test data generation and verification tasks. The procedure dynamically combines constraint propagation and simplex solver calls in an iterating process that is coordinated by synchronization conditions. These conditions are based on simple integer cutting planes and domain pruning. The originality of this procedure over existing approaches comes from its ability to deal with non-linear constructs such as disjunctions (including conditional constraints and reification), variables multiplication, Euclidian division and modulo. This paper also presents the results of an experiment on the application of this procedure to the verification of a software component of the Traffic Alert and Collision Avoidance System (TCAS). For this experiment, we used a publicly available implementation of the main component of TCAS that is responsible for alerts and resolution advisories issuance. Our results show that using CP for automated verification of software units is viable and efficient.

The rest of the paper is organized as follows. The following section presents the TCAS. Section 3 describes the constraint solving procedure based on linear relaxation and cooperation between a finite domain solver and a linear programming solver. Section 4 presents our implementation while section 5 discusses of experimental results and related works. Finally, section 6 concludes the paper and draws some perspectives.

2 TCAS software verification

The TCAS is an on-board aircraft conflict detection and resolution embedded system. The system is intended to alert the pilot to the presence of nearby aircraft that pose a mid-air collision threat and to propose maneuvers so as to resolve these potential conflicts. In cases of collision threats, TCAS estimates the time remaining until the two aircrafts reach the closest point of approach (CPA) and presents two main levels of alert. As shown on Fig.1, when an intruder aircraft enters a protected zone, the TCAS issues a Traffic Advisory (TA) to inform the pilot of potential threat. If the danger of collision increases then a Resolution Advisory (RA) is issued, providing the pilot with a proposed maneuver that is likely to solve the conflict. The RAs issued by TCAS are currently restricted to the vertical plane only (either climb or descend) and their computation depends on time-to-go to CPA, range and altitude tracks of the intruder[2]. Any TCAS implementation must be certified under level B of the DO-178B standard[3] (DO-178B 1992). According to the standard, certifying TCAS requires to show that all the executable statements and decisions of the source code has been executed at least once during the testing phase. In addition, any non-executable statement must be removed from the source code because these statements do not trace back to any software requirements and do not perform any required functionality.

[2] Future generations of TCAS may propose three-dimensional escape maneuvers
[3] The standard classifies systems with 5 criticality levels: from the highest critical level A to the least critical E

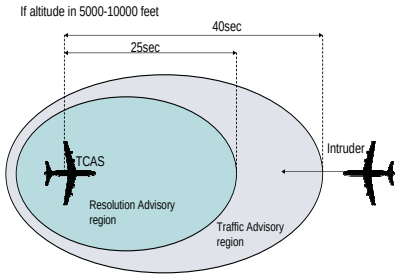


Figure 1 TCAS alarms

There are many implementations of TCAS, but obtaining the source code of these proprietary implementations is not easy. From the Software-artifact Infrastructure Repository (Do *et al.* 2005), it is possible to download a C component, called `tcas.c`, of a preliminary version of TCAS. This freely and publicly available component is responsible for the Resolution Advisories issuance. The component is (modestly) made up of 173 lines of C code. The code contains nested conditionals, logical operators, type definitions, macros and function calls, but no floating-point variables, pointers or dynamically allocated structures. Fig.2 shows the call graph of the program while Fig.3 shows the code of the highest-level function `Alt_sep_test` which computes the RAs. This function takes 14 global variables as input, including

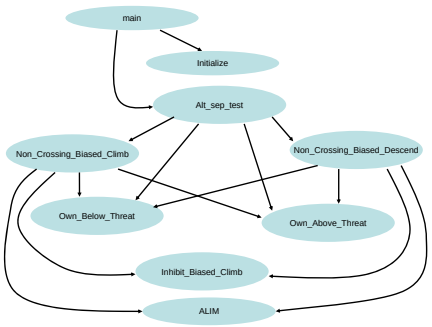


Figure 2 Call graph of `tcas.c`

`Own_Tracked_Alt` the altitude of the TCAS equipped airplane, `Other_Tracked_Alt` the altitude of the "threat", `Positive_RA_Alt_Thresh` an adequate separation threshold, `Up_Separation` the estimated separation altitude resulting from an upward maneuver and `Down_Separation` the estimated separation altitude resulting from a downward maneuver. The documentation associated to this code indicates that lines 11-12 of Fig.3 is non-executable[4], showing that this implementation is indeed not compliant with DO-178B. Any TCAS implementation must verify safety properties that come from the aircraft anti-collision theory, as presented in the TCAS II version 7 manual (TCAS II). For the considered

[4]The implementation under test considers only a single threat, hence condition of line 11-12 cannot be satisfied


```
          int alt_sep_test()
          {
1.        bool enabled, tcas_equipped, intent_not_known;
2.        bool need_upward_RA, need_downward_RA;
3.        int alt_sep;

4.        enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
                    && (Cur_Vertical_Sep > MAXALTDIFF);
5.        tcas_equipped = (Other_Capability == TCAS_TA);
6.        intent_not_known = (Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT);

7.        alt_sep = UNRESOLVED;

8.        if (enabled && ((tcas_equipped && intent_not_known) ||!tcas_equipped))
          {
9.             need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
10.            need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
11.            if (need_upward_RA && need_downward_RA)
                    // unreachable: Own_Below_Threat and Own_Above_Threat can't be both true
12.                 alt_sep = UNRESOLVED;
13.            else if (need_upward_RA)
14.                 alt_sep = UPWARD_RA;
15.                 else if (need_downward_RA)
16.                      alt_sep = DOWNWARD_RA;
                         else
17.                      alt_sep = UNRESOLVED;
          }

18.       return alt_sep;
          }
```

Figure 3 Function alt_sep_test from `tcas.c`

component, several properties referring to the possibility of issuing either an upward or a downward RA have been previously formalized in (Livadas *et al.* 1999) and (Coen-Porisini *et al.* 2001). Tab.1 shows the five double properties formalized in (Coen-Porisini *et al.* 2001). For example, property P1b says that if

Table 1 Safety properties for tcas.c

Num.	Property	Explanation	ACSL specification
P1a	Safe advisory selection	An downward RA is never issued when an downward maneuver does not produce an adequate separation	`assumes Up_Separation >= Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh; ensures result != need_Downward_RA;`
P1b	Safe advisory selection	An upward RA is never issued when an upward maneuver does not produce an adequate separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation >= Positive_RA_Alt_Tresh; ensures result != need_Upward_RA;`
P2a	Best advisory selection	A downward RA is never issued when neither climb or descend maneuvers produce adequate separation and a downward maneuver produces less separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh && Down_Separation < Up_Separation; ensures result != need_Downward_RA;`
P2b	Best advisory selection	An upward RA is never issued when neither climb or descend maneuvers produce adequate separation and an upward maneuver produces less separation	`assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh && Down_Separation > Up_Separation; ensures result != need_Upward_RA;`
P3a	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	`assumes Up_Separation ≥ Positive_RA_Alt_Tresh && Down_Separation ≥ Positive_RA_Alt_Tresh && Own_Tracked_Alt > Other_Tracked_Alt; ensures result != need_Downward_RA;`
P3b	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	`assumes Up_Separation ≥ Positive_RA_Alt_Tresh && Down_Separation ≥ Positive_RA_Alt_Tresh && Own_Tracked_Alt < Other_Tracked_Alt; ensures result != need_Upward_RA;`
P4a	No crossing advisory selection	A crossing RA is never issued	`assumes Own_Tracked_Alt > Other_Tracked_Alt; ensures result != need_Downward_RA;`
P4b	No crossing advisory selection	A crossing RA is never issued	`assumes Own_Tracked_Alt < Other_Tracked_Alt; ensures result != need_Upward_RA;`
P5a	Optimal advisory selection	The RA that produces less separation is never issued	`assumes Down_Separation < Up_Separation; ensures result != need_Downward_RA;`
P5b	Optimal advisory selection	The RA that produces less separation is never issued	`assumes Down_Separation > Up_Separation; ensures result != need_Upward_RA;`

an upward maneuver does not produce an adequate separation while an downward maneuver does, such as in Fig.4, then an upward RA should not been produced. These properties, among others, are currently

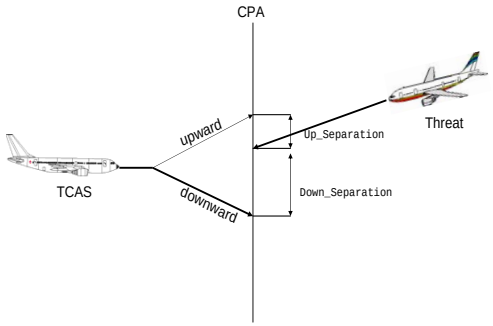


Figure 4 Resolution Advisories

verified through manual code reviews. A challenge in this area is to provide automated dependable tools that generate test data, check the conformance of a given implementation with safety properties, and show that any statement of the source code is executable. Several experimental tools based on software model-checking or static analysis exist for these tasks, but according to our knowledge, none of them is used in operational context on a regular basis. Most of them are still inefficient to deal with non-linear constraints resulting from constraint disjunction, variable multiplication, division, large array accesses and updates, pointer aliasing and so on. Moreover, proving that a given statement is indeed executable cannot be easily achieved by standard static analysis, as these analyses compute over-approximations of the program states. On the contrary, automatic test case generation tools can produce test inputs able to activate some selected statements of a program, proving so that they are indeed executable.

3 Constraint generation and solving

Our approach is based on a two-stage process. The former stage, called *constraint generation*, aims at extracting a constraint program from a given test objective and the program under test. The second stage, called *constraint solving*, aims at solving the resulting constraint system in order either to generate test data or to verify the test objective. Constraint generation is now well-documented and our approach has already been discussed in other papers: (Gotlieb *et al.* 2000) presents a constraint generation approach for imperative programs with integer computations, (Botella *et al.* 2006) discusses how to deal with floating-point computations while (Gotlieb *et al.* 2007) explains how pointer aliasing can be tackled. On the contrary, the constraint solving stage that relies on constraint propagation, linear programming and linear relaxations, and labeling has been hardly documented. As this paper focusses mainly on constraint solving (Sec. 3.3), we just briefly recall constraint generation (Sec. 3.2) after having presented the scope and the notations of our approach (Sec. 3.1).

3.1 Scope and notations

In this paper, we restrict our presentation to the fragment of the C language required to implement critical sections of programs. The TCAS implementation of our case study is made of arithmetical and logical operations over bounded integers, conditionals and function calls, but there are no loops [5], arrays or pointers. Hence, in this paper we confine our presentation to a small subset of the C language, although the approach can deal with unbounded loops (Denmat *et al.* CP 2007) and other constructions (Gotlieb 2009).

[5] TCAS is a realtime system that executes its loop-free software at each cycle


```
int foo(int x, int y)
    int z, u ;
1.  if (x * y < 4) u = 5 ; else u = 100 ;
2.  if (u ≤ 8) y = x + u ; else y = x − u;
3.  z = x * y ;
4.  assert (z > −2500)
```

Figure 5 Program foo

In the rest of the paper, variables of the program under test are noted with lower-case letter while logicial variable are noted with upper-case letters. Any logical variable X is a finite domain variable (FD variable) with $\underline{X}$ denotating the lower bound of its domain while $\overline{X}$ denotating its upper bound. Arithmetical constraints include constraints built over logical variables and operators such as $+, -, *, /, mod, etc.$ while high-level constraints include conditional constraints (noted $c_1 \longrightarrow c_2$), constraints that handle disjunctions and function calls. Note that $c_1 \longrightarrow c_2$ denotes only half of a logical implication (e.g. $\neg c_2 \longrightarrow \neg c_1$ is not a logical deduction of this constraint).

3.2 Constraint generation

3.2.1 Principle

The idea is based on the generation of a constraint program that represents the whole program under test and the test objective. While other approaches explore one by one each source-code path (Clarke *et al.* 2003, Chaki *et al.* 2004, Godefroid *et al.* 2005, Sen *et al.* 2005, Williams *et al.* 2005), we choose to explore dynamically all the possible alternatives of the program under test during constraint solving. For example, for each conditional of the program, there are two possible subpaths depending on whether the decision of the conditional is true or not. In standard constraint generation approaches, a choice is made and one of the two subpaths is explored. If this yields a contradiction, the process backtracks until a satisfactory path is found. Contradictions come from path infeasibility: a path is infeasible if and only if the decisions that govern its execution are unsatisfiable. It is worth noticing that programs that contain n conditionals have $O(2^n)$ source-code paths in the worst case, among which some may be infeasible, and then standard depth-first search approaches face a combinatorial explosion problem. To leverage this problem, our constraint generation approach implements constraint-based exploration, meaning that each conditional is considered as a constraint that may suspend when there is no subpath alternative to privilege. Instead of using choice points to represent conditionals, our approach keeps implicit disjunctions under the form of specific conditional constraint operators that apply deduction rules to determine whether each disjunct is compatible with the rest of the constraints or not. The constraint model we build represents the whole program under test and when it is considered for constraint solving, it generates a disjunctive constraint system.

3.2.2 Example

To illustrate this generation, we show the constraint generation stage on the simple example program shown in Fig.5. The program takes two 32-bit signed integer variables as inputs and defines two local variables z and u. At line 4, an assertion checks the value of z. Suppose we want to automatically verify the *test objective* which consists in verifying this assertion, i.e. to check that any state that reaches this assertion admit a value of z strictly greater than -2500. In this example, the assertion may be invalidated as there exists a test input (values for x and y) that can produce a state where $z = -2500$ (see Sec.3.3.7). The constraint generation step of our method produces the following conditional constraint system, where ite stands for if–then–else: $X, Y \in -2^{31}..2^{31}-1, Z = 0, U = 0, ite(X * Y < 4, U_1 = 5, U_1 = 100), ite(U_1 \leq 8, Y_1 = X + U_1, Y_1 = X - U_1), Z_1 = X * Y_1, Z_1 \leq -2500$. This generation uses a renaming scheme, called *Static Single Assignment (SSA)*, to tackle the problem of destructive assignment (Brandis *et al.* 1994). When the same variable x is assigned twice in the tested program, two logical

instances of the variable X_1 and X_2 are created to properly handle this situation in the constraint program (see (Gotlieb *et al.* 07) for details).

3.2.3 Test objectives

As said before, *test objectives* represent target assertions to verify in the code. Formaly speaking, if SC is the constraint system representing the whole program under test and C is an assertion to check, then searching solutions of $SC \wedge \neg C$ answers this verification problem. If one gets that $sol(SC \wedge \neg C) = \emptyset$ then assertion C is verified as it holds for any executions of the program. On the contrary, if one gets a solution s then s can be converted into a test input that violate the assertion C. The easiest way to introduce test objectives in the constraint system is therefore, 1) to constrain the execution flow in the program for reaching the assertion, 2) to falsify the assertion in order to find counter-example or to prove it. These goals are handled through a bi-directional process called *reification*, which constrains the truth value of a given constraint. For example, $R \Leftrightarrow X > Y$ associates the reification variable R to the truth value of constraint $X > Y$. By constraining R, one can specify whether the constraint has to be truth or false. On the contrary, if all the values of domains for X and Y satisfy $X > Y$ (resp. $X \leq Y$) then $R = 1$ (resp. $R = 0$). Thanks to this process, the control flow of the program is represented with a boolean constraint network over reification variables. When conditionals are nested, implication between the reification variables of the associated decisions represents the control flow (Gotlieb 09). Accordingly, assertions are reified and violating an assertion is easily specified by setting its reification variable to 0.

3.3 Constraint solving

Our constraint solving procedure is based on the cooperation of several techniques, namely constraint propagation with bound consistency, linear programming and linear relaxation, and labeling. Each of these techniques has been abundantly described in the literature (Handbook of Constraint Programming 2006) and then we focus on the combination of these techniques to validate the TCAS implementation. The constraint solver cooperation is documented in section 3.3.1, while sections 3.3.2 to 3.3.6 present linear relaxations of several standard operators. Finally, sec. 3.3.7 presents our approach to explore the search space.

3.3.1 The cooperation process

Our constraint solving procedure uses a main propagation queue that manages each constraint in turn. There are two priority levels that can be associated to a constraint. The highest priority level is given to the arithmetical constraints that can prune very early and efficiently the search space. High-level constraints such as conditionals or function calls are tackled with the lowest priority level as they implement costly entailment checks. Our implementation uses a synchronous communication process between two dedicated solvers: a finite domains constraint propagation solver (FD solver) and a simplex over the rationals (LP solver). Each time an arithmetical constraint is encountered in the propagation queue, two constraints are posted: the arithmetical constraint itself is posted in the FD solver and a linear relaxation over rationals of the arithmetical constraint is posted in the LP solver. This relaxation is computed by using current bounds of domains and then it is called a *dynamic linear relaxation (DLR)* to underline its iterative computation during the constraint solving process. We explain below with more details how these relaxations can be computed. Each variable of the original problem comes in two flavors: a FD variable for the FD solver and a rational variable for the Linear Programming solver. For example, an integer type declaration such as `int x` yields a domain constraint in the FD solver $X_{FD} \in -2^{31}..2^{31} - 1$ and a linear constraint in the LP solver $-2^{31} \leq X_Q \leq 2^{31} - 1$ where X_{FD} denotes an integer variable and X_Q denotes a rational variable.

We selected an LP implementation over the rationals in order to preserve correctness. Using a more efficient implementation based on floating-point computations would have been desirable, but our goal is to preserve the semantics of integer computations, and floating-point computations in LP are sometimes unsafe to preserve all the real solutions. In the conclusion section of the paper, we discuss the possible use

Constraint	Dynamic Linear Relaxation
$X \in a..b$	$a \leq X \leq b$
$Z = F_{lin}(X, Y, ...)$	$Z = F_{lin}(X, Y, ...)$
$Z = X * Y$	$\begin{cases} Z - X.\underline{Y} - \underline{X}.Y + \underline{XY} & \geq 0 \\ X.\overline{Y} - Z - \underline{X}.\overline{Y} + \underline{X}.Y & \geq 0 \\ \overline{X}.Y - \overline{X}.\underline{Y} - Z + X.\underline{Y} & \geq 0 \\ \overline{XY} - \overline{X}.Y - X.\overline{Y} + Z & \geq 0 \end{cases}$
$R \Leftrightarrow F(X, Y, ...) \leq 0$	$\begin{cases} F(X, Y, ...) \leq \overline{F}.(1 - R)), \\ (1 - F(X, Y, ...) \leq (1 - \underline{F}).R) \end{cases}$
$ite(c, F1, F2)$	$\begin{cases} c \longrightarrow F1 \\ \neg c \longrightarrow F2 \\ \neg(c \wedge F1) \longrightarrow (\neg c \wedge F2) \\ \neg(\neg c \wedge F2) \longrightarrow (c \wedge F1) \\ join_{dom}(F1, F2) \\ join_{lin}(F1, F2) \end{cases}$

Figure 6 Dynamic Linear relaxations (DLRs)

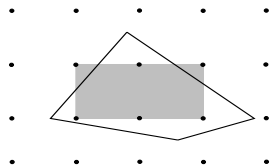


Figure 7 The grey box contains all the integer points inside the polyhedron

of safe Linear Programming implementations, such as (Neumaier *et al.* 2004), to preserve the semantics. So, having relaxed the constraint system over integer variables into a linear problem over rationals permits to check for satisfiability with more efficiency. Indeed, the LP solver considers the constraint system as a whole while the FD solver checks for satisfiability by combining only local tests of each constraint.

Note also that some nonlinear constraints over integers can be handled with relaxations. We provide in Fig.6 the formulas we used for relaxing constraints having variable multiplication, reification, conditionals, etc. Several strategies can be followed to make both solvers cooperate. We tried several heuristics and kept the one that gave the best results in average. The idea is to call each solver in turn (without interleaving) to benefit from their combined efficiency. Starting with the FD solver, a constraint propagation step with bound-filtering consistency is launched. Upon fixpoint, the DLRs are then computed using the current bounds of variable and the LP solver is called by optimizing the bounds of each variable. When the relaxed linear problem does not contain any solutions, it means that the original (possibly nonlinear) problem is unsatisfiable. This property comes from the fact that correctness was preserve by the use of relaxations over rationals. On the contrary, when the relaxed linear problem contains solutions, one can still benefit from the simplex to prune the variation domains of variables. The idea is to project the current polyhedron over each variable and comparing the resulting domain with the current domain of variable. This algorithm performs two calls to the simplex algorithm per variable appearing in the linear relaxation, one call for each bound. Then, it prunes the current domain of variable by updating its (rational) bounds. Finally, integer rounding shaves the domain to fit with integer solutions only. For example, if a call to the simplex returns $\frac{7}{2}$ as an upper bound for X, then variable X has to be lower or equal to 3. As a result, the constraint solving process computes an integer bounding box that includes all the integer solutions of the linear relaxed problem. Note however that the box may also include integer points that are no part of the polyhedron, as shown in Fig.7. Pruning the domain of variables awakes FD

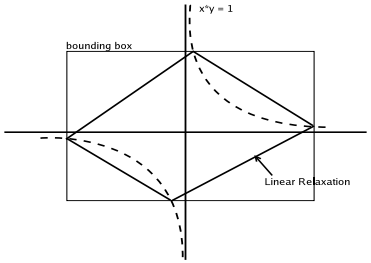


Figure 8 Relaxation of multiplication

constraints still suspended and constraint propgation can be re-launched to get better over-approximation. This iterating process is repeated until a fixpoint is reached for both solvers. At this point, a labeling procedure is launched that can possibly awake constraint propagation and DLRs computations.

A drawback of this approach is the possible slow convergence of the iterating process. However any approach that make at least two solvers cooperate faces a similar problem. In practice, we did not observe any slow convergence phenomenon in our experiments until now.

3.3.2 DLR of multiplication

The formula of Fig.6 for multiplication directly follows from the four following trivial inequalities (McCormick 1976):

$$\left\{\begin{array}{l}(X-\underline{X})(Y-\underline{Y})\geq 0\\(X-\underline{X})(\overline{Y}-Y)\geq 0\\(\overline{X}-X)(Y-\underline{Y})\geq 0\\(\overline{X}-X)(\overline{Y}-Y)\geq 0\\Z=X*Y\end{array}\right. \Rightarrow \left\{\begin{array}{ll}Z-X.\underline{Y}-\underline{X}.Y+\underline{XY} & \geq 0\\X.\overline{Y}-Z-\underline{X}.\overline{Y}+\underline{X}.Y & \geq 0\\\overline{X}.Y-\overline{X}.\underline{Y}-Z+X.\underline{Y} & \geq 0\\\overline{XY}-\overline{X}.Y-X.\overline{Y}+Z & \geq 0\end{array}\right.$$

Fig.8 shows a slice of the relaxation where $Z=1$. Here, the rectangle corresponds to the bounding box of variables X and Y, the dashed curve represents exactly $X*Y=1$, while the four solid lines correspond to the inequalities of the relaxation.

3.3.3 DLR of division, modulo and logical operators

Expressions built on division and modulo can be treated with similar linear relaxation reformulation. The constraint $Q=A\ div\ B$ where div denotes the Euclidian division rewrites to

$$(B*Q\leq A)\wedge(A<B*(Q+1))\wedge(B\neq 0)$$

Similarly, $R=A\ mod\ B$ where mod denotes the Euclidian remainder rewrites to

$$(R=A-B*Q)\wedge(0\leq R<B)$$

We applied the same principle for logical operators by studying the semantics of operators of the C programming language. The constraint $Z=X\ \&\&\ Y$ where && denotes the "logical and" operator rewrites to

$$Z=X*Y\ mod\ 2$$

The constraint $Z=X\ ||\ Y$ where || denotes "logical or" rewrites to

$$(Z=(X+Y-X*Y)\ mod\ 2)\wedge$$
$$(Z\geq X\ mod\ 2)\wedge(Z\geq Y\ mod\ 2)$$

while $Y=\tilde{}X$ where ˜denotes "logical not" rewrites[6] to

$$(X_0\in 0..1)\wedge(X*X_0=X)\wedge(Y=1-X_0)$$

3.3.4 Handling reification

Reified constraints appear in the constraint system as the result of control flow specification (reaching a specific location) or assertion violation. Consider the reified constraint $R\Leftrightarrow C$ where R is the reification variable. Without any loss of generality, let suppose that C if of the form $F(X)\leq 0$ and the function F is bounded, i.e. there exist $\underline{F}$ and $\overline{F}$ such that $\forall X\in D_X\ \underline{F}\leq F(X)\leq\overline{F}$. Then $R\Leftrightarrow F(X)\leq 0$ rewrites to the conjunction

$$(F(X)\leq\overline{F}.(1-R))\wedge(1-F(X)\leq(1-\underline{F}).R)$$

For example, consider the constraint $R\Leftrightarrow X\leq Y$, then $F(X,Y)=X-Y$, $\underline{F}=\underline{X}-\overline{Y}$, $\overline{F}=\overline{X}-\underline{Y}$, and the reified constraint rewrites to

$$(X-Y-(\overline{X}-\underline{Y})*(1-R)\leq 0)\wedge$$
$$(Y-X+1-(\overline{Y}-\underline{X}+1)*R\leq 0)$$

Note that these inequations are interpreted over the rationals. Hence, the boolean variable R is interpreted as a rational over the continuous set $[0,1]$ and then the set of solutions of these constraints (over-) approximates only the solutions over integers.

3.3.5 Handling conditionals

We use a special operator called ite for handling conditionals. This operator is implemented as a global constraint, meanning that it can be awoken when the variation domain of at least one of its variables changes. Once awoken, the algorithm of this constraint tries to prove that one of the two disjuncts is unsatisfiable with the rest of the constraints and, thus, replace the overall disjunction by the other disjunct. When this reasoning fails, the union of domains is computed.

Constraint ite is modeled using four guarded constraints. The former two directly come from the operational semantics of a conditional in a C program: $c\longrightarrow F1$, $\neg c\longrightarrow F2$. They are used to propagate forward control flow throughout the constraint system. On the contrary, the latter two implement backward reasoning: $\neg(c\wedge F1)\longrightarrow(\neg c\wedge F2)$ and $\neg(\neg c\wedge F2)\longrightarrow(c\wedge F1)$. Roughly speaking, ite represents exclusive disjunction between two disjuncts and these four guarded constraints come from this declarative view. It is worth noticing that the key behind guarded constraint implementation is constraint entailment. A constraint entailment test permits to evaluate the guard and it can be implemented by adding the negation of the guard to the rest of the constraint store and check whether the resulting system is unsatisfiable. When the system is still satisfiable (or at least partially consistent), the negated constraint must be removed from the store and other constraint entailment tests can be performed. Note also that other ite or function call operators can be nested in a given guarded constraint. To alleviate the potential combinatorial explosion of constraint entailment checks, we set up a bound on the depth of search within the guards. For example, if the bound is set to 1, then no other ite or function calls operators of $F2$ will be unfolded when the guard $\neg c\wedge F2$ is explored. In practice, we set up the bound to 2, meaning that every couple of conditionals of the program were explored for finding inconsistencies and then to prune the search space. When both cases within an ite are explored without success, then two join operators are posted: a join operator on finite domains to merge the results for each FD variable, and a join on polyhedra.

In the example of Fig.5, suppose for the sake of clarity that the domains of each variable are initailly restricted to $-1000..1000$. From the first conditional constraint $ite(X*Y<4,U_1=5,U_1=100)$, we get that the domain of U_1 is trivially pruned to $5..100$. From the second conditional constraint $ite(U_1\leq 8,Y_1=X+U_1,Y_1=X-U_1)$, the domain of X in the then-part is pruned to $-1000..995$ and the domain of Y_1 to $-995..1000$ as $U_1=5$ in this case. In the else-part, the domain of X is pruned to $-900..1000$ and the domain of Y_1 to $-1000..900$ as $U_1=100$. Perfoming the domain join on both

[6] In C, any non-null integer value is understood as true

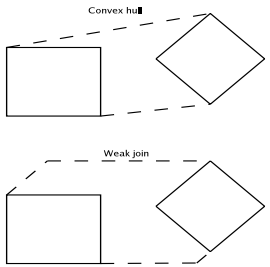


Figure 9 Weak Join vs Convex Hull

domains leaves the domains of X and Y_1 unchanged. On the contrary, the polyhedral join offers more precise results. From the second conditional, we get that $-100 \leq Y_1 - X \leq 5$. The smallest and then more desirable linear relaxation of the union of of two subsets of linear inequations is the *the convex hull* of the two corresponding polyhedra. Unfortunately, computing the convex hull of two polyhedra (when they are given under the form of inequations) is exponential in the number of dimensions. By noticing that any over-approximation of the convex hull is suitable in our context, we proposed in (Denmat et al. ISSRE 07) to use the *weak-join* operator, originally proposed in the Abstract Interpretation community (Sankarananyan *et al.* 2006). Roughly speaking, the weak-join of two sets of inequations consists in 1) enlarging the first polyhedron without changing the slope of the lines until it encloses the second polyhedron ; 2) enlarging the second polyhedron in the same way ; 3) returning the intersection of these two new sets of inequations. Fig.9 shows the difference between the convex hull and the weak join of two polyhedra E and F. Formally, let $S = E \cup F$ be the set of inequations that appear in E or in F. Let suppose that each inequation in S is of the form $A_i.X \leq b_i$ where A_i is a vector of n coefficients, X is a vector of n variables and b_i is a rational number. For each inequation in S do

$$\begin{aligned} e &= maximize(A_i.X, E) \\ f &= maximize(A_i.X, F) \\ c &= max(e, f) \end{aligned}$$

$maximize(A_i.X, E)$ denotes a call to the simplex algorithm that computes the maximum value of expression $A_i.X$ under the linear constraints E. Then, $join_{lin}(E, F) = \{A_i.X \leq c\}_{i \in 1..|S|}$. Based on the simplex, this algorithm performs well in practice.

3.3.6 *Handling function calls*

In the TCAS implementation, dealing efficiently with functions is important as there are many function calls, some of them being irrelevant to prove a given safety-critical property. In our framework, function calls are handled with a special operator called rel_call, that can be awoken on domain prunings, as any other constraint. The idea is to use *lazy evaluation* of function calls by unfolding calls only when necessary. Initially, function calls are simply ignored, and the consistency of the constraint store is evaluated without the constraints of the function calls. If the store is still partially consistent, the constraints issued from one callee are introduced into the propagarion queue and again, the consistency is evaluated. This process is iterated until no more function calls can be considered. This strategy is motivated by our will to minimize the number of constraints used to prove a given assertion. There are many other heuristics to introduce constraints of the callee function, but we did not yet pushed further this analysis. We implemented this strategy by a simple breadth first strategy over the call tree of the function under test.

3.3.7 *Labeling*

As our TCAS problem includes non-linear constraints over bounded integers, the final step of the resolution process may include a labeling phase to exhibit a solution or prove there is no solution. However as the number of variables and the variation domain of each variable are both large (TCAS takes 14 global 32-bits variables as input), resorting to enumeration to show unsatisfiability is usually prohibitive. Hence, most of the hard work has to be performed during constraint propagation. Linear relaxation of non-linear constraint was introduced in our framework to answer this problem, but as it computes over-approximations, there remain cases for wich one resorts to labelling. For selecting variable and value to enumerate first, we explored several available heuristics such as first-fail, domain splitting or most-constrained first. We also implemented two other simple labeling heuristics: iterative domain splitting and random labeling. Iterative domain splitting selects a variable X from a (static) input list and a value v in the domain of this variable, makes a non-deterministic choice between $X = v$, $X > v$ and $X < v$ and iterates over these processes until no more variable remains unassigned. Random labeling does the same except that the value v is chosen at random, using a uniform probability distribution over the domain. Randomization is interesting in the context of test data generation as it introduces uncertainty in the way test data are selected. In average, we found that these two heuristics performed well on the various distinct requests for TCAS. Coming back to the example of Fig.5, using this labeling step, we found that there are counter-examples to the assertion of line 4. For example $(X = 50, Y = 1)$ is a solution to the (nonlinear) constraint system: $X, Y \in -2^{31}..2^{31} - 1, U_1 \in 5..100, ite(X * Y < 4, U_1 = 5, U_1 = 100), ite(U_1 \leq 8, Y_1 = X + U_1, Y_1 = X - U_1), -100 \leq Y_1 - X \leq 5, Z_1 = X * Y_1, Z_1 \leq -2500$ and then using these values as a test data for program foo yields assertion violation on line 4. It is worth noticing this counter-example has been found without making any choice in the conditionals during initial propagation. Replacing the assertion of line 4 by **assert** $(z > -3000)$ in the source code of program foo yields "fail" when one tries to find counter-examples, which indicates that the assertion is satisfied by any state.

4 Implementation

The procedure described in this paper has been implemented following the architecture shown in Fig.10. The procedure takes a C file as input, optionally annotated with pre/post conditions, assertions or reach

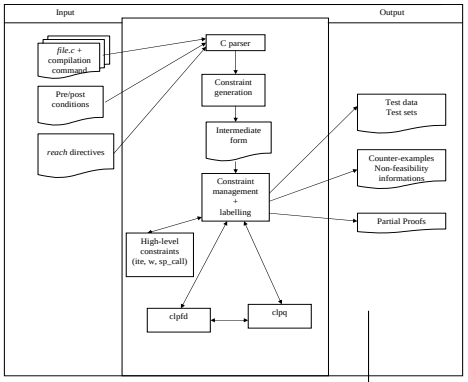


Figure 10 Implementation

directives. A directive "reach" specifies a location to reach within the code. Parsing the C file builds an abstract syntax tree and a symbol table. Then, using a normalization and points-to analysis, the syntax tree

is transformed into SSA form and then, a constraint intermediate form is produced. From there, constraint solving is launched according to some parameterization through an evaluator component. The solving procedure is based on the combination of two existing solvers through a high-level propagation queue: the `clpfd` library of SICStus Prolog which implements finite domains constraint solving (Carlsson *et al.* 1997) ; and the `clpq` library that implements a linear programming solver based on Fourier's elimination and simplex over the rationals (`clp(q,r)` Manual). When a solution is found, it is reported to the user as a test data that satisfies the test objective (reach a given location, violate an assertion or find a counter-example to a post-condition). When the solver outputs "fail", meaning that the constraint system is inconsistent, then this indication is also reported to the user. It indicates that the test objective is unsatisfiable or the assertion is verified.

The constraint solving procedure is mainly developed in Prolog ($\sim$10 KLOC) and C ($\sim$0.3 KLOC). The internal components include a backtrackable C parser written with the Definite Clause Grammar of Prolog, a SSA form generator based on the single-pass generation algorithm of (Brandis *et al.* 1994), a constraint intermediate form generator and parser, a library of high-level constraints that implements most of the C operations (conditionals, loops, logical operators, function call operator, memory operations,...). To make both constraint solvers cooperates, we exploited the SICStus global constraint interface to define constraints that awakes constraint propagation over FD. One weakness of this approach is that it delegates constraint management to the system and does not allow the order in which the constraints are considered in the queue to be modified easily. Note also that SICStus `clpfd` combines indistinctly two levels of filtering: *domain–consistency* and *interval–consistency*. Moreover, we encountered some limit problems with `clpfd` as the value of an FD variable is represented on less than 32-bits. Another problem concerns the semantics of integer computations within the constraint solver that does not mimic the semantics of integer computations in a C program, which implements the so-called *wrapping effect*. In fact, arithmetic operators in C programs implement arithmetic modulo $8, 16, 32$ or 64 bits and some programs may use this effect either conscientiously or not. For example, a statement such as `z = x+y` where `x,y,z` are 32-bits integers should be interpreted as $z = (x + y)\ mod(2^{32})$. However using this formulation would have been catastrophic in our context as 32-bit integers cannot be represented in the SICStus FD constraint solver[7] and bound-consistency on modulo operator is weak in general. As a consequence, our approach implicitly rejects any state of the program that exploits the wrapping effect.

5 Results and analysis

We conducted several experiments on a small software component of the TCAS to evaluate the capabilities of our constraint procedure to serve as an aid for testing and verification. Firstly, we evaluated structural test data generation for the coverage of the all_decisions criterion. Covering this criterion is mandatory in the context of a DO-178B B level certification. On an Intel Core Duo 2.4GHz clocked PC with 2GB of RAM, Euclide generated a test set covering all the executable decisions of the `tcas` program in 16.9 seconds, including time spent garbage collecting, stack shifting, or in system calls. It also showed that the decision of line 11-12 of Fig.3 was non executable in less than 0.2 second. Secondly, we evaluate automatic program verification on the safety properties of Tab.1. Results are shown in Tab.2.

Finding counter-examples to safety properties is usually dramatic. Hopefully the software component we used probably corresponds to a preliminary version and it has never been used in operational conditions.

Surprisingly, we found that properties P2B, P3A and P5B were not proved w.r.t. the implementation and counter-examples were exhibited. These counter-examples satisfy the preconditions but they invalidate the postconditions when they are submitted to the implementation. So, they are realistic counter-examples. All the material of these experiments, including the test data corresponding to counter-examples, is available online[8]. We executed the implementation with test data and dynamically checked that properties P2B, P3A and P5B were indeed violated. The counter-examples to properties P5B were not reported in other papers (Coen-Porisini *et al.* 2001,Clarke *et al.* 2003, Chaki *et al.* 2004). Moreover, these counter-examples

[7]SICStus Prolog version 3.12.8
[8]`www.irisa.fr/lande/gotlieb/resources.html`

Table 2 Verification of safety properties

Num	Results	Time (sec.)	Mem. (MB)
P1a	Property proved	0.7	4.6
P1b	Property proved	0.7	4.6
P2a	Property proved	0.6	4.6
P2b	Counter-example found	0.7	4.6
P3a	Counter-example found	5.4	6.3
P3b	Property proved	1.2	4.6
P4a	Counter-example found	6.8	6.9
P4b	Counter-example found	2.7	5.9
P5a	Property proved	0.6	4.6
P5b	Counter-example found	1.0	4.6

and proofs were obtained quickly (all the counter-examples and proofs are generated in less than 20s on our standard machine) which is encouraging for a future comparison with other dedicated tools.

6 Related work

Automatic program verification is a fundamental topic that was recently revamped, due to the considerable improvments in SAT– and SMT– solving[9]. Software model-checkers such as Save (Coen-Porisini *et al.* 2001), Blast (Henzinger *et al.* 2003), Magic (Chaki *et al.* 2003) or Cbmc (Clarke *et al.* 2003) routinely find counter-examples to temporal properties over C programs. These tools explore the paths of a bounded model by decomposing path constraints into SAT-formula. These formula, extracted from C expressions, are checked for satisfiability or unsatisfiability (Brummayer *et al.* 2007). Some of them also exploit *predicate abstraction* and counter-example refinement to boost the exploration. Our constraint solving contrasts with SAT-based or SMT-based model-checkers as it does not abstract the program and does not generate spurious counter-example paths. In particular it builds a high-level constraint model of C program by capturing error-free semantics without considering a boolean abstraction of the program structure. In addition, our approach exploit linear programming relaxations to solve nonlinear constraint systems, something which is currently outside the scope of SMT-solvers. Building a constraint solving procedure that makes an FD solver and an LP solver cooperate is not new. In 1995, Beringer and De Backer proposed a global constraint that captures the linear constraints of the problem. This approach was generalized in (Milano *et al.* 2002) and (Hooker *et al.* 2000). Linear relaxations of nonlinear constraints is also an old idea that dates back to the seventies (Balas 1985, McCormick 1976) and has been extended to CP by Refalo with the idea of "tight cooperation" (Refalo 1999). Recently, (Lebbah *et al.* 2005) proposed to use similar principles to deal with quadratic constraints over continuous domains. Although there are some similarities, our constraint solving approach distinguishes because it addresses specifically disjunctive non-linear constraint systems over bounded integers and preserves the correctness of results even when relaxing non-linear constraint. Our approach has also similarities with the Collavizza and Rueher (Collavizza *et al.* 2006) approach that calls several constraint solvers in sequence. Recently, they showed that their CPBPV implementation could outperform usual software model-checkers on classical benchmarks (Collavizza *et al.* 2008). CPBPV is based on deductive constraint programming techniques that staticaly combines SAT solving, linear programming and constraint propagation. However, more experimental work still need to be performed to confirm these results, obtained on a restricted subset of academic programs. According to our knowledge, our application of rational linear relaxations to software verification is original.

Automatic test data generation based on constraint propagation has been early explored in (Bicevskis *et al.* 1979) and (DeMillo *et al.* 1991) and (Offut *et al.* 1999). In this latter work, the *dynamic domain*

[9]SMT: Satisfiability Modulo Theories

reduction procedure which implements constraint propagation with bound-consistency was proposed. PathCrawler (Williams *et al.* 2005) is a recent path-oriented structural test data generators based on FD constraint solving. It exploits the constraint library Colibri developed by Bruno Marre within the CEA that implements several powerful prunings techniques such as difference logics and congruence relations in addition to bound-filtering. Dart (Godefroid *et al.* 2005) and CUTE (Sen 2005) are two other popular path-oriented test data generators based on Linear Programming over floating-point variables (lpsolve) and concrete execution. Unlike these approaches, our constraint solving procedure preserves correctness by using LP over rationals. It may be less efficient, but preserving correctness is essential in a context where properties over programs must be verified and not only tested.

7 Conclusion

In this paper, we presented a constraint solving procedure dedicated to the verification of safety-critical properties of C programs. As a first validation step, our solver was used on a freely available software component extracted from a real application (TCAS), for which safety-critical properties have been defined. It found that a complete test set covering all the executable decisions of the program could be obtained in less than 20 sec of CPU time. Our approach could also prove that some of the safety-critical properties were satisfied while other were invalidated, in a few seconds. Hence, these preliminary results show that using Constraint Programming techniques for property verification is viable and efficient. However, both foundational and applied research works still need to be undertaken in order to address more realistic implementations that contain hundreds of thousand lines of code. Methods to integrate new verification tools in the development chain should also be proposed. In our framework, we exploited an existing FD solver that manages its own propagation queue and implements its own filtering algorithms. We forecast the development of a dedicated finite domain solver based on bound-consistency filtering that could be used to check real-sized integer computations. Modelling accurately the wrapping effect would permit to find bugs related to integer representation which is outside the scope of current test data generators. Another line of research concerns the integration of a safe LP implementation over floating-point computations (Neumaier *et al.* 2004) instead of using less efficient implementation over rationals. As preserving correctness is essential when one wants to prove safety properties, this requires safe over-approximations of the solution set to be computed.

Acknowledgements

I am very grateful to Tristan Denmat who investigated the role of Abstract Interpretation in the ideas presented here. In particular, he provided us with the *weak join* idea that comes from this community. Many thanks to David Delmas from Airbus Industries and the anonymous referees for their careful reading of preliminary versions of the paper.

References

(Balas 1985) E. Balas. Disjunctive Programming and a hierarchy of relaxations for discrete optimization problems. SIAL Journal of Alg. Disc. Meth. Vol. 6, No. 3, July 1985

(Bicevskis *et al.* 1979)
J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. SMOTL - a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5(1):60–66, January 1979.

(Botella *et al.* 2006)
B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.

(Brandis *et al.* 1994)
M.M. Brandis and H. Mőssenbőck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. ACM TOPLAS, 16(6):pp 1684-1698, Nov. 1994.

(Brummayer *et al.* 2007)
R. Brummayer, A. Biere. C32SAT: Checking C Expressions. In Proc. 19th Intl. Conf. on Computer Aided Verification (CAV'07), LNCS vol. 4590.

(Carlsson *et al.* 1997)
M. Carlsson, G. Ottosson, and B. Carlson. An open–ended finite domain constraint solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.

(Chaki *et al.* 2004)
S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

(Clarke *et al.* 2003)
E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proc. of ASP-DAC'03*, pages 308–311, Jan. 2003.

(`clp(q,r)` Manual)
C. Holzbaur. *OFAI clp(q,r) Manual*. Austrian Research Institute for Artificial Intelligence, Vienna, 1.3.3 edition.

(Coen-Porisini *et al.* 2001)
A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*, pages 142–150, Vienna, Austria, September 2001. ACM.

(Collavizza *et al.* 2006)
H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 182–196, 2006.

(Collavizza *et al.* 2008)
H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.

(DeMillo *et al.* 1991)
R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

(Denmat *et al.* CP 2007)
T. Denmat, A. Gotlieb, and M. Ducasse. An abstract interpretation based combinator for modeling while loops in constraint programming. In *Proceedings of Principles and Practices of Constraint Programming (CP'07)*, Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.

(Denmat *et al.* ISSRE 2007)
T. Denmat, A. Gotlieb, and M. Ducasse. Improving Constraint-Based Testing with Dynamic Linear Relaxations. In 18th IEEE International Symposium on Software Reliability Engineering (ISSRE' 2007), Trollhttan, Sweden, Nov. 2007.

(Do *et al.* 2005)
H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

(DO-178B 1992)
DO-178B / ED-12B Software Considerations in Airborn Systems and Equipment Certification, RTCA and EUROCAE 1992

(Godefroid *et al.* 2005)
P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223, 2005.

(Gotlieb *et al.* 2000)
A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proceedings of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.

(Gotlieb *et al.* 2007)
A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, Sep. 2007.

(Gotlieb 2009)
A. Gotlieb. Euclide: A constraint-based testing platform for critical c programs. In 2th International Conference on Software Testing, Validation and Verification (ICST'09) Denver, CO, Apr. 2009..

(Handbook of Constraint Programming 2006)
Handbook of Constraint Programming Edited by Francesca Rossi, Peter van Beek, Toby Walsh Elsevier, 2006

(Henzinger *et al.* 2003)
T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of 10th Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.

(Hooker *et al.* 2000)
J. Hooker, G. O. Erlender, S. Thorsteinsson, and H.-J. Kim A scheme for unifying optimization and constraint satisfaction methods The Knowledge Engineering Review Volume 15 , Issue 1 (March 2000) pp 11 - 30

(Lebbah *et al.* 2005
Y. Lebbah, C. Michel, and M. Rueher A rigorous global filtering algorithm for quadratic constraints CONSTRAINTS Journal, 10(1),pp.47-65 ,January 2005.

(Livadas *et al.* 1999)
C. Livadas, J. Lygeros, and N.A. Lynch. High-level modeling and analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

(McCormick 1976)

G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part 1 - convex underestimating problems. *Mathematical Programming*, 10:147–175, 1976.

(Milano *et al.* 2002)
M. Milano, G. Ottosson, P. Refalo, E. S. Thorsteinsson The Role of Integer Programming Techniques in Constraint Programming's Global Constraints INFORMS JOURNAL ON COMPUTING, Vol. 14, No. 4, Fall 2002, pp. 387-402

(Neumaier *et al.* 2004)
A. Neumaier, O. Shcherbina Safe bounds in linear and mixed-integer linear programming Mathematical Programming: Series A and B archive, Vol. 99, Issue 2 (March 2004)

(Offut 1999)
J.A. Offut, Z. Jin, and Pan J. The dynamic domain reduction procedure for test data generation. *Software–Practice and Experience*, 29(2):167–193, 1999.

(Randimbivololona 2001)
F. Randimbivololona Orientations in Verification Engineering of Avionics Software Informatics, 2001, pp 131-137, LNCS

(Refalo 1999)
P. Refalo. Tight cooperation and its application in piecewise linear optimization. In Proc. of CP'99, Alexandria, Virginia, Oct. 1999.

(Sankaranarayanan *et al.* 2006)
S. Sankaranarayanan, M. A. Colòn, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *Proc. of VMCAI'06*, pages 115–125, 2006.

(Sen *et al.* 2005)
K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/FSE-13*, pages 263–272. ACM Press, 2005.

(TCAS II)
U.S. Department of transportation Federal Aviation Administration. *Introduction to TCAS II - version 7*, Nov. 2000.

(Williams *et al.* 2005)
N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *In Proc. Dependable Computing - EDCC'05*, pages 281–292, 2005.

Le travail relaté dans cet article a été présenté lors du workshop CT4ATC/ATM[2] (*Constraint Technology for Air Traffic Control/Air Traffic Management*) organisé par Pierre Flener, Justin Pearson et Marc Bourgois, durant la conférence INO (*7th EuroControl Innovative Research Workshop & Exhibition*).

Ce chapitre clos la troisième partie de notre manuscrit concernant les applications du test à base de contraintes auxquelles nous avons contribuées.

[2] `http://www.it.uu.se/research/group/astra/ATM-CT/`

Part IV

Bilan et Perspectives

Chapter 10

Bilan

Depuis une quinzaine d'années maintenant, nos recherches ont porté sur le test à base de contraintes. L'idée poursuivie et expérimentée sous différents angles a consisté à dresser un pont entre d'un côté, la *Programmation par Contraintes*, et de l'autre côté le *Test Logiciel*. Ce pont repose sur des fondations solides telles que des transformations systématiques de programmes en modèles à contraintes, l'exploitation de techniques d'abstraction pour la résolution exacte de systèmes de contraintes, l'utilisation similaire de méthodes approchées dans les deux domaines, la coïncidence de notions essentielles telles que insatisfiabilité d'un système de contraintes et infaisabilité d'un chemin d'exécution ou d'un comportement. Au travers différentes collaborations, nos contributions ont porté à la fois sur ces aspects fondamentaux, mais également sur leurs applications. La transformation systématique des constructions de programmes impératifs en systèmes de contraintes nous a permis de démontrer l'importance de la résolution de contraintes pour la génération automatique de données de test. Nous nous sommes intéressés au traitement de constructions considérées comme étant difficiles à gérer pour n'importe quelle approche de vérification de programmes. Le traitement des boucles sous forme d'un combinateur spécial, fondé sur l'Interprétation Abstraite a constitué pour nous une contribution essentielle [Gotlieb 00b, Denmat 07a]. La résolution de contraintes sur les nombres à virgule flottante [Botella 06,Carlier 11b] et sur les entiers modulaires [Gotlieb 10a] sont également des sujets de première importance pour lesquels les approches à base de contraintes que nous avons développées ont un rôle certain à jouer. Les deux modèles à contraintes, développés pour traiter le problème de la synonimie due aux pointeurs, ont permis de lever un verrou quant à la génération de données de test orientées-but [Gotlieb 07, Charreteur 09]. Nous avons exploré le lien entre *Contraintes* et *Abstractions* en proposant d'utiliser certains calculs sur les domaines abstraits tels que définis en Interprétation Abstraite, pour enrichir la résolution de contraintes [Denmat 07b, Gotlieb 09d]. La réalisation du logiciel Euclide, avec son résolveur de contraintes qui marie propagation de contraintes sur les intervalles et les polyèdres [Gotlieb 09a], ainsi que son usage pour vérifier des propriétés pour un système critique réel (i.e.,

Traffic anti-Collision Alert System) [Gotlieb 09b] sont des résultats importants de la thématique *Test à Base de Contraintes*. Enfin, l'utilisation de propriétés de symétrie des programmes impératifs pour servir d'oracles partiels [Gotlieb 03a] est un sujet qui a fait florès grâce à la popularisation du test métamorphique [Gotlieb 03b].

Les travaux que nous avons menés sur la construction de modèles à contraintes pour la vérification de programmes, et le travail acharné d'une poignée d'étudiants de grande valeur, ont conduit à la réalisation de plusieurs prototypes de recherche :

1. InKa [Gotlieb 98, Gotlieb 00b, Gotlieb 06b, Gotlieb 07, Charreteur 09] pour la génération automatique de données de test pour programmes C, prototype de recherche puis outil pré-commercial développé lors de notre formation initiale chez Dassault Electronique ;

2. FPSE [Botella 06, Carlier 11b] pour l'exécution symbolique de calculs flottants conformes à l'IEEE-754, prototype de recherche coréalisé avec Bernard Botella du CEA ;

3. Taupo [Denmat 07b,Denmat 08] pour la vérification de propriétés en utilisant des techniques d'interprétation abstraite, prototype de recherche réalisé lors de la thèse de doctorat de Tristan Denmat ;

4. Genetta-CC(FD)-PRT [Petit 07a,Petit 07b,Petit 08,Gotlieb 10b] pour la génération probabiliste de données de test, prototype de recherche réalisé lors de la thèse de doctorat de de Matthieu Petit ;

5. EUCLIDE [Gotlieb 09a, Gotlieb 09b] pour la vérification de programmes C critiques, prototype de recherche réalisé alors que nous étions à l'INRIA Rennes ;

6. JAUT [Charreteur 10a] pour la génération automatique de cas de test pour le Bytecode Java, prototype de recherche réalisé lors de la thèse de Florence Charreteur.

La plupart de ces prototypes sont déposés sous licence libre et sont disponible en ligne.
Ces travaux autour du test à base de contraintes ont été pour nous une occasion unique de rencontrer des étudiants formidables et des collègues précieux, au travers de différentes collaborations. Mais, il reste beaucoup à faire afin que le test à base de contraintes soit reconnu comme une technique importante de génération de tests et de vérification de programmes.

Chapter 11

Perspectives

Nous évoquons ici quatre perspectives de notre travail qui s'inscrivent dans le cadre du *Test à Base de Contraintes* mais en débordent, pour certaines, le champ d'application traditionnel.

Avec la thèse de Mickaël Delahaye, qui sera soutenue en octobre 2011, co-encadrée avec Bernard Botella, nous nous sommes intéressés à la généralisation de chemins infaisables en exécution symbolique dynamique. Les outils de génération automatique de données de test fondés sur l'exécution symbolique dynamique, tels que DART [Godefroid 05], PathCrawler [Williams 05] et CUTE [Sen 05], perdent souvent beaucoup de temps à démontrer que certains systèmes de contraintes sont insatisfiables. En effet, ces systèmes correspondent à des chemins infaisables, qui sont nombreux dans les programmes impératifs [Yates 89]. Nous avons proposé une approche qui tire parti de la donnée d'un chemin infaisable, pour bâtir automatiquement par généralisation, une famille de chemin infaisables [Delahaye 10]. Notre technique se fonde d'abord sur la recherche d'un noyau minimal insatisfiable du système de contraintes, en utilisant un algorithme non-intrusif tel que *"quickexplain"* [Junker 04], et ensuite sur la recherche de chemins du programme ayant la même source d'insatisfiabilité. Cette connaissance, une fois acquise par généralisation d'un chemin infaisable initial, est capitalisée sous forme d'un automate de chemins infaisables qui représente une famille potentiellement infinie de chemins. Nous avons exploré plusieurs variantes de cette approche en utilisant des résolveurs de contraintes tels que Colibri, Eclipse IC et Z3 [De Moura 08b], et des algorithmes de calculs de noyaux insatisfiables intrusifs. Nous nous sommes également penchés sur l'utilisation efficace de cet automate en génération automatique de données de tests par exécution symbolique dynamique. Nos résultats sont en cours de publication [Delahaye 11].

En collaboration avec Catherine Dubois et Matthieu Carlier, nous avons proposé un premier modèle à contraintes pour les programmes fonctionnels. Ce modèle s'appuie sur des combinateurs qui modélisent le choix conditionnel, le "pattern matching", l'appel de fonction, y compris récursif. Il est utilisé pour générer des cas de tests à partir de propriétés de programmes prenant la forme pré-condition/post-

condition [Carlier 10], en offrant une couverture structurelle de la pré-condition. Nous avons récemment étendu ce modèle afin qu'il traite également les appels de fonctions d'ordre supérieur et lancé une campagne d'expérimentations afin de démontrer l'intérêt de cette approche [Carlier 11a]. Ce travail vise ainsi à explorer le potentiel d'un modèle issu de la *Programmation par Contraintes* pour valider automatiquement des programmes écrits dans un langage fonctionnel.

Les programmes à contraintes eux-mêmes sont de plus en plus utilisés dans les systèmes critiques. Par exemple, les programmes à contraintes sont utilisés dans le commerce électronique [Holland 05], le contrôle et la gestion du trafic aérien [Flener 07, Junker 08], ou encore dans le développement et la validation de logiciels critiques [Collavizza 08, Gotlieb 09b]. Ainsi, il devient important de s'intéresser à la validation systématique de ces programmes au travers de méthodes de test logiciel. Avec la thèse de Nadjib Lazaar, qui sera soutenue à la fin de l'année 2011, et en collaboration avec Yahia Lebbah, nous avons proposé une méthode de test pour les programmes à contraintes qui consiste à utiliser un premier modèle déclaratif du problème comme oracle de test pour un modèle optimisé, construit par raffinements successifs [Lazaar 10b]. Cette méthode de test repose sur la donnée de relations de conformité valables aussi bien pour les problèmes de satisfaction que les problèmes d'optimisation. Nous nous sommes également intéressés à la localisation de fautes dans les programmes à contraintes [Lazaar 10a] et leur correction automatique [Lazaar 11]. Ces travaux sont très prometteurs et constituent selon nous une première percée dans le domaine de la validation automatique des programmes à contraintes.

Enfin, avec la thèse de Aymeric Hervieu, démarrée en 2010 et co-encadrée avec Benoit Baudry, nous nous intéressons à l'utilisation des contraintes pour la génération de configurations de test à partir d'un modèle de variabilité. Un tel modèle peut être utilisé pour représenter des lignes de produits logiciels ou des systèmes à base de composants logiciels. En effet, la variabilité dans les systèmes peut être capturée par un "feature model" qui couvre implicitement un ensemble, potentiellement immense, de configurations valides d'un système. Le problème de la sélection des configurations à tester devient alors essentiel car il n'est pas question de construire toutes les configurations valides d'un système. Une approche bien établie maintenant consiste à choisir un ensemble de configurations garantissant que chaque paire de valeurs des "feature" seront présentes dans au moins une des configurations à tester. Cette forme de test s'appelle le test combinatoire. Nous avons proposé un premier modèle à contraintes qui représente le "feature model" en utilisant des contraintes globales dédiées. Ce modèle à contraintes est utilisé dans une approche de génération de tests combinatoires pour sélectionner les configurations valides du système à tester. Cette approche présente le double intérêt de minimiser le nombre de configurations pour couvrir un critère de test sélectionné et d'être implantée comme un algorithme "anytime", c'est à dire permettant d'être interrompue à tout instant afin d'obtenir une solution approchée [Hervieu 11].

Part V

Annexe : Curriculum Vitae

Arnaud Gotlieb

INRIA Rennes Bretagne Atlantique
Campus Beaulieu 35042 Rennes Cedex

Born Jul. 23th, 1971 in Metz, France
16 rue du Linon
35720 Pleugueneuc
Pacs, 2 children

Tel: +33 (0)2 99 84 75 76
Fax: +33 (0)2 99 84 71 71
Arnaud.Gotlieb@inria.fr

http://www.irisa.fr/lande/gotlieb/

Current professional situation

Research scientist in the INRIA Celtique project-team. My research interests are centered around software testing automation and how constraint programming and constraint solving techniques can help the software testing process. My work focuses on automatic test data generation for C and Java embedded programs, constraint-based testing, software testing theories, variability testing, statistical testing and constraint reasoning in structural testing. I participated to the design and development of several constraint solving engines targeted to the testing of critical embedded programs.

Industrial and research experience

2002-now / Research scientist -- INRIA Rennes Bretagne Atlantique

- Main architect of several constraint solvers dedicated to automatic test data generation for C and Java (Euclide, FPSE, PRT,... details on www.irisa.fr/lande/gotlieb)

- Co-Author of more than thirty international publications in Research conferences and journals (Papers:41, Cites/paper: 13.5, h-index: 11, Citations: 556 – Source: Harzing's Publish&Perish, Jan. 2011)

- Scientific coordinator the CAVERN project (2008-2011), funded by the ANR SESUR 2007 programme, exploring the use of Constraints and Abstractions in Program Verification.
Partners: INRIA Celtique, IBM Ilog Lab, CEA List, University of Nice-Sophia Antipolis

- Member of the **RNTL CAT (2005-2008) and ANR U3CAT (2008-2012) projects**, developing constraint solving tools to address the verification problem of critical embedded C code.
Partners: CEA List, INRIA Proval, Dassault Aviation, Airbus Industries, ...

- Leader of the **GENETTA project (2004-2007),** funded by the Brittany region. Aims at using probabilistic constraint reasoning to perform automatic statistical structural testing.

- In 2006, expert consultant for the **AUTOTEST project**, funded by European Space Agency (ESA), under contract with LEIRIOS (now SmarTesting) -- Statistical Testing for Space software

- Co-leader of the **CHANNEL project (2004-2006),** funded by Egide under the PAI ALLIANCE, exploring automatic test data generation for security problems. In collaboration with Andy King from University of Kent (UK).

- Member of the **ACI V3F project (2004-2007),** developing constraint solving tools for floating-point computations in critical software.
Partners: LIFC, INRIA Coprin, Vertecs and Lande teams, CEA List laboratory

- Member of **RNTL CASTLES project (2003-2006)**, developing an automated certification environment for the Java Card platform. Partners: INRIA Everest and Lande teams, OBERTHUR CARD SYSTEMS, AQL

1999-2002 / Software Engineer -- Thales Airborne Systems – 78851 Elancourt – FRANCE (Ex DASSAULT ELECTRONIQUE)

- Leader of the INKA Project, developing an automated software test data generation tool for C/C++, based on Constraint Programming techniques. Validation realized on a part of the embedded software systems BCE of the ABE of RAFALE military aircraft.

- Development and deployment of the "software quality tools" offer in Thales A.S. (coding rules verification, code complexity measurement tools)

- Operational specifications of the ADA to C++ translator of test model DEVISOR (unit testing tool used to test embedded software systems of M-2000 and RAFALE military aircrafts)

1998 – 1999 Military service – Commission Armées – Jeunesse, in the French Air Army

1995 – 1998 CIFRE Fellow -- DASSAULT ELECTRONIQUE quai Marcel Dassault, Saint Cloud

- Research work on automatic test data generation with Constraint Logic Programming techniques.

- Cycle detection and optimization in **INTERLOG**: a constraint solver over continuous domains.

Education

PhD, Computer Science with high honors (félicitations jury),
University of Nice – Sophia Antipolis, France, Jan. 2000

Thesis entitled : "*Automatic test data generation with Constraint Logic Programming*"

M.Sc Mathematics with honors (mention Bien)**,**
University of Nice – Sophia Antipolis, France, 1995

B.Sc Mathematics with honors,
University of Nice – Sophia Antipolis, France, 1993

Scientific animation and administrative tasks

- **Co-president of the MTVV** group of the CNRS GDR-GPL (Resp. Yves Ledru) since its creation in 2008
- **Member of the « Comité de Sélection** » **UFR Nantes**, Université de Nantes 2009 et 2011
- **Member of the « Comité de Sélection** » **INSA de Rennes**, 2010 et 2011
- **Member of the « Comité de Sélection** » **IUT Orsay** 2010

- **Examiner in the PhD thesis jury** of Severine Colin (2005),Patricia Mouy (2007),Matthieu Carlier (2009), Nicolas Berger (2010), and external reviewer for two International PhD thesis (Australian Swinburne University and University of Sevilla in Spain)

- **Member of the INRIA's Commission des Développements Technologiques** (CDT, 2011)
- **Member of the INRIA's GROLO** Working Group on Software Tools evaluation (2009)
- **Member of the Thomson-CSF International Committee (CETs**) for the recommendation of unit testing and source code analysis tools (2000)

- **PC member** of the conferences IEEE ISSRE'04, QSIC 06-11, TAP 08-11, IEEE ICST 08-10, workshops STEV'07-08, ACM RT'07, CSTVA 06-11, VAST'10, national conferences JFPC 04-10
- **Main organizer of the three editions of the CSTVA workshop (2006, 2010, 2011)** – Constraints in Software Testing, Verification and Analysis. Co-located with IEEE ICST.
- **Publicity chair** of ICLP'04 and ISSRE'04, **Session chair** in IEEE ICST 2008-2010
- **Reviewer** for journals IEEE TSE (2010), ACM TOPLAS (2009), STVR (2005-2007), SPE, KBS, SQJ,TSI,KER and international conferences ISSRE, SAS, TACAS, ICST, TESTCOM, WLPE, Bytecode, ...
- IEEE member
- Various expertise for ANR, "Conseil Régional de Franche-Comté", EITICA Technology transfer in the Acquitaine region, Ile de France (DimLSC)

Supervision of 5 PhD students, 3 Post-docs and others

- PhD Thesis of Mickael Delahaye (2007-now) on static analysis techniques for automatic test data generation in C programs. Co-advising with B. Botella from CEA and Thomas Jensen (Habilitated). I am involved in the supervision of the thesis at 50%. In this work, we proposed to utilize infeasible path detection to improve the automatic test inputs generation [Delahaye, Botella, Gotlieb ICST'10].

- PhD Thesis of Nadjib Lazaar (2008-now) on testing constraint programs. Co-advising with Thomas Jensen (Habilitated). I am involved in the supervision of the thesis at 100%. In this work, we proposed a first approach for testing constraint programs [Lazaar, Gotlieb, Lebbah CP'10], localizing faults in constraint programs [Lazaar Gotlieb Lebbah ICTAI'10] and automatic correction of constraint programs [Lazaar Gotlieb Lebbah ICST'11].

- PhD Thesis of Florence Charreteur (Defense on 9 March 2010). "Modélisation par contraintes de programmes en bytecode java pour la génération automatique de tests", University of Rennes, Co-advising with Thomas Jensen (Habilitated). I was involved in the supervision of the thesis at 100%. In this work, we proposed a constraint model for constraint-based reasoning on Bytecode programs [Charreteur Gotlieb ISSRE'10]. This model handles dynamic data structures and it is based on a memory model that was published in [Charreteur Botella Gotlieb JSS 2009].

Since late 2009, Florence works for DGA in Bruz, as a Research engineer.

- PhD Thesis of Matthieu Petit (Defense on 4 Jul. 2008). "Using probabilistic choice constraint for statistical structural testing", University of Rennes, Co-advising with Thomas Jensen (Habilitated). I was involved in the supervision of the thesis at 100%. In this work, we proposed to use probabilistic choice operators [Petit Gotlieb CP'07] to reason on imperative programs [Gotlieb Petit JSS'10].

Since 2008, Matthieu is postdoc at the Danish University of Roskilde.

- PhD Thesis of Tristan Denmat (Defense on 5 June 2008). "Contraintes et abstractions pour la generation automatique de données de test", University of Rennes, Co-advising with Mireille Ducassé (Habilitated). I was involved in the supervision of the thesis at 50%. In this work, we proposed to use abstract domain computations in constraint combinators [Denmat Gotlieb Ducassé CP'07] to perform test input generation [Denmat Gotlieb Ducassé ISSRE'07]. We also explore pointer analyses for handling pointer aliasing in test input generation and program verification [Gotlieb Botella Denmat IST'07].

Since 2008, Tristan works as a software engineer for various software companies (currently ATOS origin).

- Post-doc. of Sandrine Gouraud (2005-2006, on RNTL CASTLES). We proposed a technique based on Constraint Handling Rules to generate test cases for the Java Card Virtual Machine [Gouraud Gotlieb PADL'06].

- Post-doc of Pierre Rousseau (2006-2007, on RNTL CAT). We worked on the early design of Euclide. In particular, we implemented the handling of the ACSL language in the tool.

- Post-doc of Matthieu Carlier (2009-2011, on ANR U3CAT). We proposed to use constraint reasoning in the testing of functional programs [Carlier Dubois Gotlieb ICSOFT'10].

- Benjamin Cama (engineer, 12months, 2008-2009, on CAVERN project). Development of a web interface for Euclide

- Nada Benduro (engineer, 7months, 2009-2010, on CAVERN project). Testing procedures for Euclide.

Students advising : more than 15 trainees over the last ten years

Teaching

2002-2011: “Validation, Verification and Test”, Course, Master level,5INFO, INSA de Rennes (30HeqTD).
2002-2010: “Constraint-based testing”, Course, Master (M2R), Université de Rennes (6HeqTD)
2006-2011: “Sofware Testing”, Course, Master level, Ecole des Mines de Nantes (18HeqTD)
2010-2011: “Compilation”, Course, Master level, 4INFO, INSA de Rennes (24HeqTD)
2010-2011: “Constraint-based testing”, ALMA Seminar, University of Nantes (3H)

2007 : TAROT Summer School “Constraint based testing” (6H)
2004 : OBERTHUR formation on “Software Testing” (6H)

Member of master student jury (M2R University of Rennes) from 2004 to 2009

Visit of Research groups and invited presentations

Nov. 2003 : Visit of the Software Assurance Laboratory (J. Bieman’s Research group)
at Colorado State University in Fort Collins – Denver
Invited presentation: *Automatic Test Data Generation with Constraint Logic Programming*

Mar. 2004 : CASSIS workshop, Marseille, France, Invited talk: Testing programs with symmetry

Nov.2005 : LIFC Seminar, Besançon, France, Invited talk :
Probabilistic Constraint Programming for Statistical Structural Testing

May 2006 : CEA Seminar - Saclay, Invited talk : *INKA: Ten years after the first idea*

Jun. 2010 : Visit of the ASTRA Research group (P. Flener, J. Pearson) at University of Uppsala, Sweden
Invited presentation: *An overview of Constraint-Based Testing*

Distinctions

- Nominated for Best paper award IEEE ISSRE 2003 “Exploiting symmetries to test programs” A. Gotlieb
- Won the prize for best poster at GDR-GPL 2010
- Gift from Microsoft Research for sponsoring the CSTVA workshop series, under the Verified Software Initiative led by Prof. T. Hoare.

Publications in International Referred Journals

M. Delahaye, B. Botella, A. Gotlieb. ***Infeasible Path Generalization in Dynamic Symbolic Execution***. Submitted for publication.to IEEE Transactions on Software Engineering

M. Carlier, C. Dubois, A. Gotlieb. ***A constraint-based approach for testing functional programs***. Submitted for publication.to the J. Wileys' Journal of Software Testing, Verification and Analysis

A. Gotlieb and M. Petit. **A uniform random test data generator for path testing**. *The Journal of Systems and Software*, 83(12):2618-2626, Dec. 2010. (IF: 1.340 Source: Elsevier, ERA2010 rank: A)

F. Charreteur, B. Botella, and A. Gotlieb. **Modelling dynamic memory management in constraint-based testing**. *The Journal of Systems and Software*, 82(11):1755–1766, Nov. 2009. Special Issue: TAIC-PART 2007 and MUTATION 2007. (IF: 1.340 Source: Elsevier, ERA2010 rank: A)

A. Gotlieb. **Tcas software verification using constraint programming**. *The Knowledge Engineering Review*, 2009. Accepted for publication. (IF: 1.143 Source: Cambridge, ERA2010 rank: B)

A. Gotlieb, T. Denmat, and B. Botella. **Goal-oriented test data generation for pointer programs**. *Information and Soft. Technol.*, 49(9-10):1030–1044, Sep. 2007. (IF: 1.821 Source: Elsevier, ERA2010 rank: B)

B. Botella, A. Gotlieb, and C. Michel. **Symbolic execution of floating-point computations**. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006. (IF: 1.632 Source: Wiley, ERA2010 rank: B)

O. Lhomme, A. Gotlieb, and M. Rueher. **Dynamic optimization of interval narrowing algorithms**. *Journal of Logic Programming*, 37:164–182, 1998. (New journal name: Theory and Practice of Logic Programming, IF: 1.467 Source: Cambridge, ERA2010 rank: A)

Publications in International Referred Conferences

N. Lazaar, A. Gotlieb, and Y. Lebbah. **A framework for the automatic correction of constraint programs**. In *4th IEEE International Conference on Software Testing, Validation and Verification (ICST'11)*, Berlin, Germany, Mar. 2011.

F. Charreteur and A. Gotlieb. **Constraint-based test input generation for java bytecode**. In Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10), San Jose, CA, USA, Nov. 2010. (Research papers selection: 43/130, acceptance rate: 33%, Source: IEEE proceedings, ERA2010 rank: A)

M. Carlier, C. Dubois, and A. Gotlieb. **Constraint reasonning in focaltest**. In 5rd International Conference on Software and Data Technologies (ICSOFT'10), Athens, Greece, Jul. 2010.
(Full paper selection: 24/266, acceptance rate: 9%, Source: ICSOFT, ERA2010 rank: B)

M. Delahaye, B. Botella, and A. Gotlieb. **Explanation-based generalization of infeasible path**. In 3rd IEEE International Conference on Software Testing, Validation and Verification (ICST'10), Paris, France, Apr. 2010. (Paper selection: 50/189, acceptance rate: 27%, Source: IEEE Proc., ERA2010 rank: C)

N. Lazaar, A. Gotlieb, and Y. Lebbah. **Fault localization in constraint programs**. In 22th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI'2010), Arras, France, Oct. 2010.
(Regular paper selection: 67/243, acceptance rate: 27.5%, Source: IEEE proceedings, ERA2010 rank: B)

N. Lazaar, A. Gotlieb, and Y. Lebbah. **On testing constraint programs**. In 16th Int. Conf. on Principles and Practices of Constraint Programming (CP'2010), St Andrews, Scotland, Sept. 2010.
(Research paper selection: 36/101, acceptance rate: 36%, Source: LNCS 6308, ERA2010 rank: A)

A. Gotlieb. **Euclide: A constraint-based testing platform for critical c programs**. In 2th IEEE International Conference on Software Testing, Validation and Verification (ICST), Denver, CO, Apr. 2009. (Paper selection: 47/140, acceptance rate: 33%, Source: IEEE Proc., ERA2010 rank: C)

A. Gotlieb and M. Petit. **Towards a theory for testing non-terminating programs**. In 33nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09), Seattle, USA, Jul. 2009. 6 pages. (Paper selection: (46+29 short)/231, acceptance rate: 32.4%, Source: IEEE Proc., ERA2010 rank: B)

A. Gotlieb and M. Petit. **Constraint reasonning in path-oriented random testing**. In 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'08), Turku, Finland, Jul. 2008. Short paper, 4 pages. (Research paper selection: (46+36 short)/236, acceptance rate: 34.7~~%, Source:~~ IEEE Proc., ERA2010 rank: B)

F. Charreteur, B. Botella, and A. Gotlieb. **Modelling dynamic memory management in constraint-based testing**. In TAIC-PART (Testing: Academic and Industrial Conference), Windsor, UK, Sep. 2007.

T. Denmat, A. Gotlieb, and M. Ducasse. **An abstract interpretation based combinator for modeling while loops in constraint programming**. In Proceedings of Principles and Practices of Constraint Programming (CP'07), Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007. (Research paper selection: 43/143, acceptance rate: 30%, Source: LNCS 4741, ERA2010 rank: A)

T. Denmat, A. Gotlieb, and M. Ducasse. **Improving constraint-based testing with dynamic linear relaxations**. In 18th IEEE International Symposium on Software Reliability Engineering (ISSRE' 2007), Trollhättan, Sweden, Nov. 2007. (Paper selection : 26/78, acceptance rate: 33%, Source: IEEE proceed., ERA2010 rank: A)

M. Petit and A. Gotlieb. **Boosting probabilistic choice operators**. In Proceedings of Principles and Practices of Constraint Programming, Springer Verlag, LNCS 4741, pages 559–573, Providence, USA, September 2007. (Research paper selection: 43/143, acceptance rate: 30%, Source: LNCS 4741, ERA2010 rank: A)

M. Petit and A. Gotlieb. **Uniform selection of feasible paths as a stochastic constraint problem**. In Proceedings of International Conference on Quality Software (QSIC'07), IEEE, Portland, USA, October 2007. (ERA2010 rank: B)

A. Gotlieb and P. Bernard. **A semi-empirical model of test quality in symmetric testing: Application to testing java card APIs**. In Sixth International Conference on Quality Software (QSIC'06), Beijing, China, Oct. 2006. (Paper selection : 50/181, acceptance rate: 28%, Source: Proceedings, ERA2010 rank: B)

A. Gotlieb, B. Botella, and M. Watel**. Inka: Ten years after the first ideas**. In 19th Int. Conf. on Soft. and Systems Eng. and their Applications (ICSSEA'06), Paris, France, Dec. 2006.

S.D. Gouraud and A. Gotlieb. **Using chrs to generate test cases for the jcvm**. In Eighth International Symposium on Practical Aspects of Declarative Languages, PADL 06, Charleston, South Carolina, January 2006. LNCS 3819 (ERA2010 rank: B)

A. Gotlieb, T. Denmat, and B. Botella. **Constraint-based test data generation in the presence of stack-directed pointers**. In 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA, Nov. 2005. 4 pages. (Paper selection : (28+35 short)/291, acceptance rate: 21.6%, Source: IEEE/ACM proceedings, ERA2010 rank: A)

A. Gotlieb, T. Denmat, and B. Botella. **Goal-oriented test data generation for programs with pointer variables**. In 29th IEEE Annual International Computer Software and Applications Conference (COMPSAC'05), pages 449–454, Edinburh, Scotland, July 2005. 6 pages. (Paper selection : 71/278, acceptance rate: 25.5%, Source: IEEE proceedings, ERA2010 rank: B)

A. Gotlieb. **Exploiting symmetries to test programs**. In IEEE International Symposium on Software Reliability and Enginering (ISSRE), Denver, CO, USA, November 2003. (Paper selection : 41/200, acceptance rate: 21%, Source: IEEE proceedings, ERA2010 rank: A)

A. Gotlieb and B. Botella. **Automated metamorphic testing**. In 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC'03), Dallas, TX, USA, November 2003. (ERA2010 rank: B)

A. Gotlieb. **Inka: An automatic software test data generator.** In Proceedings of DAta Systems In Aerospace (DASIA 2001), Eurospace, The Association of European Space Industry, Nice, France, May 2001.

A. Gotlieb, B. Botella, and M. Rueher. **A clp framework for computing structural test data**. In Proceedings of Computational Logic (CL'2000), LNAI 1891, pages 399–413, London, UK, July 2000. (Paper selection: 86/176, acceptance rate: 49%, Source: LNAI 1861.)

A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'98), pages 53–62, 1998. Also in Software Engineering Notes 23:2, Mar. 1998 (Paper selection: 16/47, acceptance rate: 34%, Source: SEN Proc., ERA2010 rank: A)

O. Lhomme, A. Gotlieb, M. Rueher, and P. Taillibert. **Boosting the interval narrowing algorithm**. In Proc. of the Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96), pages 378–392, Bonn, Germany, Sep. 1996. MIT Press. (Paper selection: 35/122, acceptance rate: 28.6%, Source: Proc., ERA2010 rank: A)

Publications in International Referred Workshops

A. Gotlieb, M. Leconte, and B. Marre. **Constraint solving on modular integers**. In Proc. of the 9th Int. Workshop on Constraint Modelling and Reformulation (ModRef'10), co-located with CP'2010, St Andrews, Scotland, Sept. 2010.

M. Petit and A. Gotlieb. Distinguish **dynamic basic blocks by structural statistical testing.** In Proc. of the 12th European Workshop on Dependable Computing, Toulouse, France, May 2009.

B. Blanc, F. Bouquet, A. Gotlieb, B. Jeannet, T. Jéron, B. Legeard, B. Marre, C. Michel, and M. Rueher. **The v3f project**. In Proc. of the 1st Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA'06), Nantes, France, Sep. 2006.

A. Gotlieb and M. Petit. **Path-oriented random testing**. In 1st ACM Int. Workshop on Random Testing (RT'06), Portland, Maine, July 2006.

T. Denmat, A. Gotlieb, and M. Ducassé. **Proving or disproving likely invariants with constraint reasoning.** In Proc.of the 15th Workshop on Logic-Based Methods in Programming Environments (WLPE'05), Sitges, SPAIN, Oct. 2005. satelite event of International Conference on Logic Programming (ICLP'2005).

M. Petit and A. Gotlieb. **An ongoing work on statistical structural testing via probabilistic concurrent constraint programming**. In SIVOES-MODEVA workshop – satelite event of Int. Symp. on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, November 2004.

Publications in National Journals

B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. **Utilisation des contraintes pour la génération automatique de cas de test structurels**. In Technique et sciences informatiques, volume 21–9 of TSI, pages 1163–1187. Hermes – Lavoisier, 2002.

A. Gotlieb, F. Calvet, and M. Rueher. **Génération automatique de cas de test : une approche programmation logique par contraintes**. In Actes des journées Génie Logiciels GL'96 publiés dans Génie Logiciel, pages 135–140, Paris, France, Nov. 1996. EC2.

Publications in National Conferences

S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams. **Constraint-based software testing**. In 1eres journées nationales du GDR-GPL, groupe de travail MTVV, Toulouse, France, Jan. 2009.

N. Lazaar, A. Gotlieb, and Y. Lebbah. **Vers une théorie du test des programmes à contraintes**. In Cinquièmes Journées Francophones de Programmation par Contraintes (JFPC'09), Amiens, France, Juin 2009.

F. Charreteur and A. Gotlieb. **Raisonnement à contraintes pour le test de bytecode java**. In Quatrièmes Journées Francophones de Programmation par Contraintes (JFPC'08), pages 11–20, Nantes, France, Juin 2008.

M. Petit and A. Gotlieb. Raisonner **et filtrer avec un choix probabiliste partiellement connu**. In Secondes Journées Francophones de Programmation par Contraintes (JFPC'06), Nimes, France, Juin 2006.

S.D. Gouraud and Gotlieb A. **Utilisation des chrs pour générer des cas de test fonctionnel pour la machine virtuelle java card**. In Premières Journées Francophones de Programmation par Contraintes (JFPC'05), Lens, France, juin 2005.

Book chapter

B. Blanc, A. Gotlieb, and C. Michel. **Constraints in software testing, verification and analysis**. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, Trends in Constraint Programming, part 7, pages 333–368. ISTE, London, UK, May 2007.

Posters and other publications

A Gotlieb. **Euclide**. In 2èmes Journées Nationales du GDR-GPL, Pau, France, Jan. 2010.
won the best poster prize.

N. Lazaar, A. Gotlieb, and Y. Lebbah. **Towards constraint-based local search for automatic test data generation**. In Poster in the 1th International Workshop on Search-based Test Data Generation, co-located with ICST 2008, Lillehammer, Norway, Apr. 2008.

M. Petit and A. Gotlieb. **Probabilistic choice operators as global constraints : application to statistical software testing**. In Poster presentation in ICLP'04, number 3132 in Springer LNCS, pages 471–472, 2004.

B. Botella and A. Gotlieb. **FPSE: Floating-Point Symbolic Execution**. INRIA-IRISA, Rennes, 2005. Documentation of a floating-point interval constraint solver.

A. Gotlieb**. Utilisation de la Programmation Logique par Contraintes pour la génération automatique de données de test**. Thèse de doctorat, Université de Nice Sophia-Antipolis, Jan 2000.

Résumé

Résumé

Ces dernières années, les recherches en matière de Test Logiciel ont conduit au développement de techniques de résolution de contraintes dédiées, dans ce qui est appelé "le test à base de contraintes". Notre approche dans ce domaine vise à explorer l'apport de la Programmation par Contraintes à la génération automatique de test pour les programmes impératifs. Nous nous sommes intéressés à la résolution de problèmes combinatoires difficiles issus de la génération de données de test, en développant des techniques de propagation de contraintes et de filtrage, adaptées au traitement des constructions des langages de programmation. Notre habilitation tente de faire une première synthèse de ce sujet au travers de cinq contributions : l'hybridation de techniques de résolution de contraintes pour la génération automatique de cas de test, la génération probabiliste de cas de test à l'aide d'opérateurs à contrainte probabiliste, les contraintes sur un modèle mémoire pour les programmes manipulant les pointeurs, la résolution de contraintes sur les expressions portant sur les nombres à virgule flottante, et le test de programmes à contraintes. Nous illustrons également ces contributions par leur application à la vérification de logiciels critiques, et dressons quelques perspectives à ces travaux.

Abstract

These last years have seen the development of several constraint solving techniques dedicated to the testing of software systems, in an area called "Constraint-Based Testing". Our approach in this research domain consists to explore how Constraint Programming techniques can help the automatic generation of tests for imperative programs. We have addressed hard combinatorial problems resulting from automatic test cases generation by developing constraint propagation and filtering techniques well-tuned for handling control and data structures of imperative programs. Our habilitation tries to establish a first synthesis of the domain through five contributions, namely hybrid constraint solving for automatic test case generation, probabilistic test generation through probabilistic constraint combinator, constraints over abstract memory models, constraint solving over floating-point expressions and testing of constraint programs. We also illustrate these contributions through their application to critical software verification, and draw several research perspectives to our work.

Bibliography

[Anand 07] Saswat Anand, Corina S. Pasareanu & Willem Visser. *JPF–SE: A Symbolic Execution Extension to Java PathFinder*. In Int. Conf. on Tools and Algo. for the Construction and Analysis of Systems (TACAS'07), April 2007.

[Antoine 94] C. Antoine, P. Baudin, J.M. Collart, J. Raguideau & A. Trotin. *Using formal methods to validate C programs*. In 5th International Symposium on Software Reliability Engineering (ISSRE'94), pages 252–258, nov 1994.

[Arcuri 09] Andrea Arcuri. *Theoretical analysis of local search in software testing*. In Proceedings of the 5th international conference on Stochastic algorithms: foundations and applications, SAGA'09, pages 156–168, 2009.

[Ball 01] Thomas Ball, Rupak Majumdar, Todd Millstein & Sriram K. Rajamani. *Automatic predicate abstraction of C programs*. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, PLDI '01, pages 203–213, 2001.

[Ball 05] Thomas Ball. *A Theory of Predicate-Complete Test Coverage and Generation*. In Formal Methods for Components and Objects, volume 3657 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin / Heidelberg, 2005.

[Ball 11] Thomas Ball, Vladimir Levin & Sriram K. Rajamani. *A decade of software model checking with SLAM*. Commun. ACM, vol. 54, pages 68–76, July 2011.

[Bardin 08] S. Bardin & P. Herrmann. *Structural Testing of Executables*. In 1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08), pages 22–31, 2008.

[Bardin 09] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher & N. Williams. *Constraint-Based Software Testing*. In 1eres journées nationales du GDR-GPL, groupe de travail MTVV, Toulouse, France, Jan. 2009.

[Barnett 11] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte & H. Venter. *Specification and Verification: The Spec# Experience*. Communications of the ACM, vol. 54, no. 6, pages 81–91, June 2011.

[Baudin 09] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy & Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[Berstel 10] B. Berstel & M. Leconte. *Using Constraints to Verify Properties of Programs*. In 2nd Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'10, 2010. Co-located with ICST'10 in Paris, April.

[Bicevskis 79] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins & E. Miller. *SMOTL - A System to Construct Samples for Data Processing Program Debugging*. IEEE Transactions on Software Engineering, vol. 5, no. 1, pages 60–66, January 1979.

[Bin 02] E. Bin, R. Emek, G. Shurek & A. Ziv. *Using a constraint satisfaction formulation and solution techniques for random test program generation*. IBM Systems Journal, vol. 41, no. 3, 2002.

[Bobot 11] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich. *Why3: Shepherd Your Herd of Provers*. In Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, Aug. 2011.

[Boldo 09] Sylvie Boldo, Jean-Christophe Filliâtre & Guillaume Melquiond. *Combining Coq and Gappa for Certifying Floating-Point Programs*. In Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics, Calculemus '09/MKM '09, pages 59–74, 2009.

[Boonstoppel 08] Peter Boonstoppel, Cristian Cadar & Dawson Engler. *RWset: Attacking path explosion in constraint-based test generation*. In Int. Conference on Tools and Algorithms for the Constructions and Analysis of Systems (TACAS'08), pages 351–366, 2008.

[Botella 05] B. Botella & A. Gotlieb. *FPSE: Floating-Point Symbolic Execution*. INRIA-IRISA, Rennes, 2005. Documentation of a floating-point interval constraint solver.

[Botella 06] B. Botella, A. Gotlieb & C. Michel. *Symbolic execution of floating-point computations*. The Software Testing, Verification and Reliability journal, vol. 16, no. 2, pages pp 97–121, June 2006.

[Botella 09] B. Botella, F. Charreteur & A. Gotlieb. *CAVERN - Delivrable 2.1 - Modèle mémoire*. Rapport technique ver. 1, 2009.

[Bougé 86] L. Bougé, N. Choquet, L. Fribourg & M.C. Gaudel. *Test Sets Generation from Algebraic Specifications Using Logic Programming*. The Journal of Systems and Software, vol. 6, pages 343–360, 1986.

[Bouquet 05] Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard & Mark Utting. *JML-Testing-Tools: A Symbolic Animator for JML Specifications Using CLP*. In Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS'05), pages 551–556, Apr 2005.

[Brillout 09] A. Brillout, D. Kroening & T. Wahl. *Mixed abstractions for floating-point arithmetic*. In Formal Methods in Computer-Aided Design, 2009. FMCAD 2009, pages 69 –76, nov. 2009.

[Burdy 05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. Leino & E. Poll. *An Overview of JML Tools and Applications*. International Journal on Software Tools for Technology Transfer, vol. 7, no. 3, pages 212–232, 2005.

[Burnim 08] Jacob Burnim & Koushik Sen. *Heuristics for Scalable Dynamic Test Generation*. In ASE'08: 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[C. Pasareanu 03] M. Dwyer C. Pasareanu & W. Visser. *Finding Feasible Abstract Counter-Examples*. International Journal on Software Tools for Technology Transfer, vol. 5, no. 1, 2003.

[Cadar 06] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill & D.R. Engler. *EXE: automatically generating inputs of death*. In Proc. of Comp. and Communications Security (CCS'06), pages 322–335, 2006.

[Canet 09] Géraud Canet, Pascal Cuoq & Benjamin Monate. *A Value Analysis for C Programs*. In Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM '09, pages 123–124, 2009.

[Carlier 10] M. Carlier, C. Dubois & A. Gotlieb. *Constraint Reasonning in FOCALTEST*. In 5rd International Conference on Software and Data Technologies (ICSOFT'10), Athens, Greece, Jul. 2010.

[Carlier 11a] M. Carlier, C. Dubois & A. Gotlieb. *A constraint-based approach for testing functional programs*. 2011. under revision.

[Carlier 11b] M. Carlier & A. Gotlieb. *Filtering by ULP Maximum*. In Proc. of the IEEE Int. Conf. on Tools for Artificial Intelligence (ICTAI'11), Nov. 2011. Short paper, 4 pages.

[Carlsson 97] M. Carlsson, G. Ottosson & B. Carlson. *An Open–Ended Finite Domain Constraint Solver*. In Proc. of Programming Languages: Implementations, Logics, and Programs, 1997.

[Carver 96] Richard H. Carver. *Testing abstract distributed programs and their implementations: A constraint-based approach*. Journal of Systems and Software, vol. 33, no. 3, pages 223–237, 1996.

[Chan 98] F.T. Chan, T.Y. Chen, S. C. Cheung, M.F. Lau & S.M. Yiu. *Application of metamorphic testing in numerical analysis*. In IASTED Conf. in Soft. Eng., pages 191–197, 1998.

[Charreteur 07] F. Charreteur, B. Botella & A. Gotlieb. *Modelling dynamic memory management in Constraint-Based Testing*. In TAIC-PART (Testing: Academic and Industrial Conference), Windsor, UK, Sep. 2007.

[Charreteur 08] F. Charreteur & A. Gotlieb. *Raisonnement à contraintes pour le test de bytecode Java*. In Quatrièmes Journées Francophones de Programmation par Contraintes (JFPC'08), pages 11–20, Nantes, France, Juin 2008.

[Charreteur 09] F. Charreteur, B. Botella & A. Gotlieb. *Modelling dynamic memory management in Constraint-Based Testing*. The Journal of Systems and Software, vol. 82, no. 11, pages 1755–1766, Nov. 2009. Special Issue: TAICPART 2007 and MUTATION 2007.

[Charreteur 10a] F. Charreteur & A. Gotlieb. *Constraint-Based Test Input Generation for Java Bytecode*. In Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10), San Jose, CA, USA, Nov. 2010.

[Charreteur 10b] Florence Charreteur. *Modélisation par Contraintes de programmes en bytecode Java pour la génération automatique de tests*. Thèse de doctorat, Université de Rennes 1, Mars 2010.

[Chen 01] T.Y. Chen, T.H. Tse & Zhiquan Zhou. *Fault-Based Testing in the Absence of an Oracle*. In IEEE Int. Comp. Soft. and App. Conf. (COMPSAC), pages 172–178, 2001.

[Choquet 86] N. Choquet. *Test Data Generation using a Prolog with Constraints*. In Proc. of the Workshop on Software Testing, pages 132–141, Banff, Canada, Jul. 1986.

[Chung 09] Insang Chung & James M. Bieman. *Generating input data structures for automated program testing*. Softw. Test. Verif. Reliab., vol. 19, pages 3–36, March 2009.

[Clarke 76] L. Clarke. *A System to Generate Test Data and Symbolically Execute Programs*. IEEE Transactions on Software Engineering, vol. 2, no. 3, pages 215–222, September 1976.

[Collavizza 07] H. Collavizza & M. Rueher. *Exploring different constraint-based modelings for program verification*. In Proc. of CP2007, LNCS 4741, pages 49–63, 2007.

[Collavizza 08] H. Collavizza, M. Rueher & P. Van Hentenryck. *CPBPV: A Constraint-Programming Framework for Bounded Program Verification*. In Proc. of CP2008, LNCS 5202, pages 327–341, 2008.

[Coppit 05] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le & Kevin Sullivan. *Software Assurance by Bounded Exhaustive Testing*. IEEE Transactions on Software Engineering, vol. 31, pages 328–339, 2005.

[Cousot 77] P. Cousot & R. Cousot. *Abstract Interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In Proceedings of Symp. on Principles of Programming Languages, pages 238–252. ACM, 1977.

[Cousot 92] Patrick Cousot & Radhia Cousot. *Abstract Interpretation Frameworks*. J. Log. Comput., vol. 2, no. 4, pages 511–547, 1992.

[Cousot 05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival. *The ASTREÉ Analyzer*. In 14th European Symposium on Programming (ESOP'05), Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, pages 21–30, Apr. 2005.

[Davis 87] E. Davis. *Constraint Propagation With Interval Labels*. Artificial Intelligence, vol. 32, pages 281–331, 1987.

[de Moura 08a] Leonardo de Moura & Nikolaj Bjørner. *Model-based Theory Combination*. Electron. Notes Theor. Comput. Sci., vol. 198, no. 2, pages 37–49, 2008.

[De Moura 08b] Leonardo De Moura & Nikolaj Bjørner. *Z3: an efficient SMT solver*. In TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, pages 337–340. Springer-Verlag, 2008.

[Degrave 09] François Degrave, Tom Schrijvers & Wim Vanhoof. *Towards a Framework for Constraint-Based Test Case Generation*. In Logic-Based Program Synthesis and Transformation, 19th International Symposium, LOPSTR 2009, Coimbra, Portugal, September 2009, Revised Selected Papers, pages 128–142, 2009.

[Delahaye 10] M. Delahaye, B. Botella & A. Gotlieb. *Explanation-based generalization of infeasible path*. In 3rd IEEE International Conference on Software Testing, Validation and Verification (ICST'10), Paris, France, Apr. 2010.

[Delahaye 11] M. Delahaye, B. Botella & A. Gotlieb. *Infeasible Path Generalization in Dynamic Symbolic Execution*. 2011. under revision.

[Delmas 09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal & Franck Védrine. *Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software*. In Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems, FMICS '09, pages 53–69, 2009.

[Delzanno 01] Giorgio Delzanno & Andreas Podelski. *Constraint-based deductive model checking*. International Journal on Software Tools for Technology Transfer (STTT), vol. 3, no. 3, pages 250–270, 2001.

[DeMillo 91] R.A. DeMillo & J.A. Offut. *Constraint-Based Automatic Test Data Generation*. IEEE Transactions on Software Engineering, vol. 17, no. 9, pages 900–910, September 1991.

[Denmat 05] T. Denmat, A. Gotlieb & M. Ducassé. *Proving or Disproving likely invariants with constraint reasoning*. In Proc.of the 15th Workshop on Logic-Based Methods in Programming Environments (WLPE'05), Sitges, SPAIN, Oct. 2005. satelite event of International Conference on Logic Programming (ICLP'2005).

[Denmat 07a] T. Denmat, A. Gotlieb & M. Ducasse. *An Abstract Interpretation Based Combinator for Modeling While Loops in Constraint Programming*. In Proceedings of Principles and Practices of Constraint Programming (CP'07), Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.

[Denmat 07b] T. Denmat, A. Gotlieb & M. Ducasse. *Improving Constraint-Based Testing with Dynamic Linear Relaxations*. In 18th IEEE International Symposium on Software Reliability Engineering (ISSRE' 2007), Trollhättan, Sweden, Nov. 2007.

[Denmat 08] Tristan Denmat. *Contraintes et abstractions pour la génération automatique de données de test*. Thèse de doctorat, Institut National des Sciences Appliquées de Rennes, Juin 2008.

[Detlefs 05] David Detlefs, Greg Nelson & James B. Saxe. *Simplify: a theorem prover for program checking*. J. ACM, vol. 52, pages 365–473, May 2005.

[Dick 93] J. Dick & A. Faivre. *Automating the Generation and Sequencing of Test Cases from Model-based Specifications*. In Proc. of the FME'03: Industrial Strength Formal Methods, LNCS 670, 1993.

[Do 05] Hyunsook Do, Sebastian G. Elbaum & Gregg Rothermel. *Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.* Empirical Software Engineering: An International Journal, vol. 10, no. 4, pages 405–435, 2005.

[Filliâtre 04] J.C. Filliâtre & C. Marché. *Multi-prover Verification of C Programs*. In 6th Int. Conf. on Formal Engineering Methods (ICFEM'04), pages 15–29, Nov. 2004.

[Flanagan 02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe & R. Stata. *Extended static checking for Java*. In Conf. on Programming Language Design and Implementation, pages 234–245. ACM, 2002.

[Flanagan 04] Cormac Flanagan. *Automatic software model checking via constraint logic.* Sci. Comput. Program., vol. 50, no. 1-3, pages 253–270, 2004.

[Flener 07] P. Flener, J. Pearson, M. Agren, Garcia-Avello C., M. Celiktin & S. Dissing. *Air-traffic complexity resolution in multi-sector planning*. Journal of Air Transport Management, vol. 13, no. 6, pages 323 – 328, 2007.

[Gerlich 10] Ralf Gerlich. *Generic and Extensible Automatic Test Data Generation for Safety Critical Software with CHR*. In Proc. of 7th Workshop on Constraint Handling Rules (CHR'10), Jul. 2010.

[Godefroid 05] P. Godefroid, N. Klarlund & K. Sen. *DART: directed automated random testing*. In Proc. of PLDI'05, pages 213–223, 2005.

[Godefroid 08a] P. Godefroid, P.de Halleux, A. Nori, S.K. Rajamani, W. Schulte, N. Tillmann & M.Y. Levin. *Automating Software Testing Using Program Analysis*. IEEE Software, vol. 25, no. 5, pages 30–37, 2008.

[Godefroid 08b] Patrice Godefroid, Michael Y. Levin & David A. Molnar. *Automated Whitebox Fuzz Testing*. In NDSS'08: Network and Distributed System Security Symposium. The Internet Society, 2008.

[Godefroid 09] Patrice Godefroid. *Software Model Checking Improving Security of a Billion Computers*. In Corina Pasareanu, editeur, Model Checking Software, volume 5578 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin / Heidelberg, 2009.

[Godefroid 10] Patrice Godefroid & Johannes Kinder. *Proving memory safety of floating-point computations by combining static and dynamic program analysis*. In Proc. of the International Symposium on Software Testing and Analysis (ISSTA'10), pages 1–12, Jul. 2010.

[Gorlick 90] M. Gorlick, C. Kesselman, D. Marotta & S. Parker. *Mockingbird: A Logical Methodology for Testing*. volume 8, pages 95–119, 1990.

[Gotlieb 98] A. Gotlieb, B. Botella & M. Rueher. *Automatic Test Data Generation Using Constraint Solving Techniques*. In Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'98), pages 53–62, 1998.

[Gotlieb 00a] A. Gotlieb. *Génération automatique de cas de test structurel avec la Programmation Logique par Contraintes*. Thèse de doctorat, Université de Nice Sophia Antipolis, January 2000.

[Gotlieb 00b] A. Gotlieb, B. Botella & M. Rueher. *A CLP Framework for Computing Structural Test Data*. In Proceedings of Computational Logic (CL'2000), LNAI 1891, pages 399–413, London, UK, July 2000.

[Gotlieb 03a] A. Gotlieb. *Exploiting Symmetries to Test Programs*. In IEEE International Symposium on Software Reliability and Enginering (ISSRE), Denver, CO, USA, November 2003.

[Gotlieb 03b] A. Gotlieb & B. Botella. *Automated Metamorphic Testing*. In 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC'03), Dallas, TX, USA, November 2003.

[Gotlieb 05a] A. Gotlieb, T. Denmat & B. Botella. *Constraint-based test data generation in the presence of stack-directed pointers*. In 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA, Nov. 2005. 4 pages.

[Gotlieb 05b] A. Gotlieb, T. Denmat & B. Botella. *Goal-oriented test data generation for programs with pointer variables*. In

29th IEEE Annual International Computer Software and Applications Conference (COMPSAC'05), pages 449–454, Edinburh, Scotland, July 2005. 6 pages.

[Gotlieb 06a] A. Gotlieb & P. Bernard. *A Semi-empirical Model of Test Quality in Symmetric Testing: Application to Testing Java Card APIs*. In Sixth International Conference on Quality Software (QSIC'06), Beijing, China, Oct. 2006.

[Gotlieb 06b] A. Gotlieb, B. Botella & M. Watel. *Inka: Ten years after the first ideas*. In 19th Int. Conf. on Soft. and Systems Eng. and their Applications (ICSSEA'06), Paris, France, Dec. 2006.

[Gotlieb 07] A. Gotlieb, T. Denmat & B. Botella. *Goal-oriented test data generation for pointer programs*. Information and Soft. Technol., vol. 49, no. 9-10, pages 1030–1044, Sep. 2007.

[Gotlieb 08] A. Gotlieb & M. Petit. *Constraint reasonning in Path-oriented Random Testing*. In 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'08), Turku, Finland, Jul. 2008. Short paper, 4 pages.

[Gotlieb 09a] A. Gotlieb. *EUCLIDE: A Constraint-Based Testing platform for critical C programs*. In 2th IEEE International Conference on Software Testing, Validation and Verification (ICST'09), Denver, CO, Apr. 2009.

[Gotlieb 09b] A. Gotlieb. *TCAS software verification using Constraint Programming*. The Knowledge Engineering Review, 2009. Accepted for publication.

[Gotlieb 09c] A. Gotlieb & M. Petit. *Towards a Theory for Testing Non-terminating Programs*. In 33nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09), Seattle, USA, Jul. 2009. 6 pages.

[Gotlieb 09d] Arnaud Gotlieb. *CAVERN - Delivrable 3.1 - Polyhedral abstractions in Constraint-Based Testing*. Rapport technique ver. 1, 2009.

[Gotlieb 10a] A. Gotlieb, M. Leconte & B. Marre. *Constraint Solving on Modular Integers*. In Proc. of the 9th Int. Workshop

on Constraint Modelling and Reformulation (ModRef'10), co-located with CP'2010, St Andrews, Scotland, Sept. 2010.

[Gotlieb 10b] A. Gotlieb & M. Petit. *A Uniform Random Test Data Generator for Path Testing*. The Journal of Systems and Software, vol. 83, no. 12, pages 2618–2626, Dec. 2010.

[Goubault 01] E. Goubault. *Static Analyses of the Precision of Floating-Point Operations*. In Static Analysis Symposium (SAS'01) and also in LNCS 2126, pages 234–245, Paris, FR, July 2001.

[Gouraud 06] S.D. Gouraud & A. Gotlieb. *Using CHRs to generate test cases for the JCVM*. In Eighth International Symposium on Practical Aspects of Declarative Languages, PADL 06, Charleston, South Carolina, January 2006. LNCS 3819.

[Gupta 98] N. Gupta, A.P. Mathur & M.L. Soffa. *Automated Test Data Generation Using An Iterative Relaxation Method*. In Foundations on Software Engineering, Orlando, FL, Nov. 1998. ACM.

[Gupta 00] Neelam Gupta, Aditya P. Mathur & Mary Lou Soffa. *Generating Test Data for Branch Coverage*. In Proc. of the Automated Software Engineering Conference, pages 219–228, 2000.

[Hari 08] Siva Kumar Sastry Hari, Vishnu Vardhan Reddy Konda, V. Kamakoti, Vivekananda M. Vedula & Kailasnath S. Maneperambil. *Automatic constraint based test generation for behavioral HDL models*. IEEE Trans. Very Large Scale Integr. Syst., vol. 16, pages 408–421, April 2008.

[Hentenryck 92] P.V. Hentenryck, H. Simonis & M. Dincbas. *Constraint satisfaction using constraint logic programming*. Artificial Intelligence, vol. 58, no. 1-3, pages 113–159, 1992.

[Hentenryck 93] P. Van Hentenryck, V. Saraswat & Y. Deville. *Design, Implementation, and Evaluation of the Constraint Language cc(FD)*. Technical Report CS-93-02, Brown University, 1993.

[Hentenryck 98] P.V. Hentenryck, V. Saraswat & Y. Deville. *Design, Implementation, and Evaluation of the Constraint Language cc(FD)*. Journal of Logic Programming, vol. 37, pages 139–164, 1998.

[Henzinger 03] T. Henzinger, R. Jhala, R. Majumdar & G. Sutre. *Software verification with Blast*. In Proc. of 10th Workshop on Model Checking of Software (SPIN), pages 235–239, 2003.

[Hervieu 11] A. Hervieu, B. Baudry & A. Gotlieb. *PACOGEN : Automatic Generation of Pairwise Test Configurations from Feature Models*. In Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE'11), Nov. 2011.

[Hoffman 91] D. Hoffman & P. Strooper. *Automated Module Testing in Prolog*. IEEE Transactions on Software Engineering, vol. 17, no. 9, Sep. 1991.

[Holland 05] Alan Holland & Barry O'Sullivan. *Robust solutions for combinatorial auctions*. In ACM Conference on Electronic Commerce (EC-2005), pages 183–192, 2005.

[Holzbaur 95] C. Holzbaur. *OEFAI clp(q,r) Manual Rev. 1.3.2*. Austrian Research Institute for Artificial Intelligence, Vienna, AU, 1995. TR-95-09.

[IEEE-754 85] IEEE-754. *Standard for Binary Floating-Point Arithmetic*. ACM SIGPLAN Notices, vol. 22, no. 2, pages 9–25, February 1985.

[Jackson 00] Daniel Jackson & Mandana Vaziri. *Finding bugs with a constraint solver*. In Proc. of ISSTA'00, pages 14–25, 2000.

[Junker 04] Ulrich Junker. *QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems*. In Proc. of the Nineteenth Nat. Conf. on Artificial Intelligence, Sixteenth Conf. on Innovative Applications of Artificial Intelligence (AAAI'04), July 25-29, 2004, San Jose, California, USA, pages 167–172, 2004.

[Junker 08] U. Junker & D. Vidal. *Air Traffic Flow Management with ILOG CP Optimizer*. In International Workshop on Constraint Programming for Air Traffic Control and Management, 2008. 7th EuroControl Innovative Research Workshop and Exhibition (INO'08).

[King 76] J.C. King. *Symbolic Execution and Program Testing*. Communications of the ACM, vol. 19, no. 7, pages 385–394, July 1976.

[Korel 90] B. Korel. *Automated Software Test Data Generation*. IEEE Transactions on Software Engineering, vol. 16, no. 8, pages 870–879, Aug. 1990.

[Lakhotia 10] Kiran Lakhotia, Nikolai Tillmann, Mark Harman & Jonathan De Halleux. *FloPSy: search-based floating point constraint solving for symbolic execution*. In Proc. of the 22nd IFIP WG 6.1 international conference on Testing software and systems, ICTSS'10, pages 142–157, 2010.

[Lazaar 10a] N. Lazaar, A. Gotlieb & Y. Lebbah. *Fault Localization in Constraint Programs*. In 22th Int. Conf. on Tools with Artificial Intelligence (ICTAI'2010), Arras, France, Oct. 2010.

[Lazaar 10b] N. Lazaar, A. Gotlieb & Y. Lebbah. *On Testing Constraint Programs*. In 16th Int. Conf. on Principles and Practices of Constraint Programming (CP'2010), St Andrews, Scotland, Sept. 2010.

[Lazaar 11] N. Lazaar, A. Gotlieb & Y. Lebbah. *A framework for the automatic correction of Constraint Programs*. In 4th IEEE International Conference on Software Testing, Validation and Verification (ICST'11), Berlin, Germany, Mar. 2011.

[Leconte 06] M. Leconte & B. Berstel. *Extending a CP Solver With Congruences as Domains for Software Verification*. In 1st Workshop on Constraints in Software Testing, Verification and Analysis, CSTVA'06, 2006. Co-located with CP'06 in Nantes, September.

[Legeard 01] B. Legeard & F. Peureux. *Generation of functional test sequences from B formal specifications - Presentation and industrial case-study*. In Proc. of the 16th IEEE Int. Conf. on Automated Software Engineering (ASE 2001), pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[Lewin 95] D. Lewin, L. Fournier, M. Levinger, E. Roytman & G. Shurek. *Constraint Satisfaction for Test Program Generation*. In IEEE International Phoenix Conference on Communication and Computers, 1995, 1995.

[Littlewood 93] Bev Littlewood & Lorenzo Strigini. *Validation of ultra-high dependability for software-based systems*. Commun. ACM, vol. 36, pages 69–80, November 1993.

[Mackworth 77] Alan K. Mackworth. *Consistency in Networks of Relations*. Artificial Intelligence, vol. 8, no. 1, pages 99–118, 1977.

[Marre 91] B. Marre. *Toward Automatic Test Data Set Selection using Algebraic Specifications and Logic Programming*. In Koichi Furukawa, editeur, Proc. of the Eight Int. Conf. on Logic Prog. (ICLP'91), pages 202–219, Paris, Jun. 1991. MIT Press.

[Marre 00] B. Marre & A. Arnould. *Test Sequences Generation from LUSTRE Descriptions: GATEL.* In Proc. of the 15th IEEE Conference on Automated Software Engineering (ASE'00). IEEE CS Press, Septembre 2000.

[Marre 05] Bruno Marre & Benjamin Blanc. *Test Selection Strategies for Lustre Descriptions in GATeL*. Electronic Notes in Theoretical Computer Science, vol. 111, pages 93 – 111, 2005.

[Marre 10] Bruno Marre & Claude Michel. *Improving the Floating Point Addition and Subtraction Constraints*. In Principles and Practice of Constraint Programming - CP'2010, volume 6308 of *LNCS*, pages 360–367. 2010.

[Merchez 01] S. Merchez, C. Lecoutre & F. Boussemart. *AbsCon: a prototype to solve CSPs with abstraction*. In Proc. Of Constraint Programming (CP'01), LNCS 2239, pages 730–744. Springer-Verlag, 2001.

[Meudec 01] C. Meudec. *ATGen: automatic test data generation using constraint logic programming and symbolic execution*. Software Testing, Verification and Reliability, vol. 11, no. 2, pages 81–96, June 2001.

[Michel 01] C. Michel, M. Rueher & Y. Lebbah. *Solving Constraints over Floating-Point Numbers*. In Proceedings of Principles and Practices of Constraint Programming (CP'01), Springer Verlag, LNCS 2239, pages 524–538, Paphos, Cyprus, November 2001.

[Michel 02] C. Michel. *Exact projection functions for floating point number constraints*. In Seventh Int. Symp. on Artificial Intelligence and MAthematics (7th AIMA), Fort Lauderdale, FL, USA, Jan. 2002.

[Miller 76] W. Miller & D. Spooner. *Automatic Generation of Floating-Point Test Data*. IEEE Transactions on Software Engineering, vol. 2, no. 3, pages 223–226, September 1976.

[Miné 04] A. Miné. *Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors*. In Proc. of the European Symp. on Programming, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[Mohammed Said Belaid 10] Michel Rueher Mohammed Said Belaid Claude Michel. *Approximating floating-point operations to verify numerical programs*. In 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN'10), ENS Lyon, France, Sep. 2010.

[Offutt 88] Jeff Offutt. *Automatic Test Data Generation*. Phd dissertation, Georgia Institute of Technology, Atlanta GA, 1988.

[Pesch 85] H. Pesch, H. Schaller, P. Schnupp & A.P. Spirk. *Test Case Generation Using Prolog*. In Proc. of the 8th Int. Conf. on Soft. Eng. (ICSE'85), pages 252–258, London, U.K., 1985.

[Petit 07a] M. Petit & A. Gotlieb. *Boosting Probabilistic Choice Operators*. In Proceedings of Principles and Practices of Constraint Programming, Springer Verlag, LNCS 4741, pages 559–573, Providence, USA, September 2007.

[Petit 07b] M. Petit & A. Gotlieb. *Uniform Selection of Feasible Paths as a Stochastic Constraint Problem*. In Proceedings of International Conference on Quality Software (QSIC'07), IEEE, Portland, USA, October 2007.

[Petit 08] Matthieu Petit. *Test statistique structurel par résolution de contraintes de choix probabiliste*. Thèse de doctorat, Université de Rennes 1, Juillet 2008.

[Podelski 00] Andreas Podelski. *Model Checking as Constraint Solving*. In Proceedings of Static Analysis Symposium (SAS'00), volume 1824 of *LNCS*, pages 22–37. Springer-Verlag, 2000.

[Pretschner 01] Alexander Pretschner & Heiko Lötzbeyer. *Model Based Testing with Constraint Logic Programming: First Results and Challenges*. In IN 2ND ICSE INT. WORKSHOP ON AUTOMATED PROGRAM ANALYSIS, TESTING, AND VERI (WAPATV'01), 2001.

[Sen 05] K. Sen, D. Marinov & G. Agha. *CUTE: a concolic unit testing engine for C*. In Proc. of ESEC/FSE-13, pages 263–272. ACM Press, 2005.

[Snelting 06] Gregor Snelting, Torsten Robschink & Jens Krinke. *Efficient path conditions in dependence graphs for software safety analysis*. ACM Trans. Softw. Eng. Methodol., vol. 15, pages 410–457, October 2006.

[Sneyers 10] Jon Sneyers, Peter Van Weert, Tom Schrijvers & Leslie De Koninck. *As time goes by: Constraint Handling Rules*. Theory and Practice of Logic Programming (TPLP), vol. 10, no. 1, pages 1–47, 2010.

[Strooper 91] P. Strooper & D. Hoffman. *Prolog Testing of C Modules*. 1991.

[Sy 01] Nguyen Tran Sy & Yves Deville. *Automatic Test Data Generation for Programs with Integer and Float Variables*. In In 16th IEEE Int. Conference on Automated Software Engineering (ASE'01), pages 3–21, 2001.

[Sy 03] Nguyen Tran Sy & Yves Deville. *Consistency techniques for interprocedural test data generation*. In Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-11, pages 108–117, 2003.

[Tang 10] Enyi Tang, Earl Barr, Xuandong Li & Zhendong Su. *Perturbing numerical calculations for statistical analysis of floating-point program (in)stability*. In Proc. of the 19th Int. Symp. on Software Testing and Analysis, ISSTA '10, pages 131–142, 2010.

[Tillmann 08] N. Tillmann & J. de Halleux. *Pex: White Box Test Generation for .NET*. In Proc. of the 2nd Int. Conf. on Tests and Proofs, LNCS 4966, pages 134–153, 2008.

[Tracey 98] N. Tracey, J. Clark & K. Mander. *Automated Program Flaw Finding using Simulated Annealing*. vol. 23, no. 2, pages 73–81, 1998.

[Truchet 10] Charlotte Truchet, Marie Pelleau & Frédéric Benhamou. *Abstract Domains for Constraint Programming, with the Example of Octagons*. In 12th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'10), Timisoara, Romania, pages 72–79, 2010.

[Tse 07] T. H. Tse, Francis C. M. Lau, W. K. Chan, Peter C. K. Liu & Colin K. F. Luk. *Testing object-oriented industrial software without precise oracles or results*. Commun. ACM, vol. 50, pages 78–85, August 2007.

[Visser 04] W. Visser, Corina S. Pasareanu & S. Khurshid. *Test input generation in Java Pathfinder*. In Proc. of ISSTA'04, 2004.

[Visvanathan 02] S. Visvanathan & N. Gupta. *Generating Test Data for Functions with Pointer Inputs*. In Proceedings of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE'02), Edinburgh, UK, September 2002.

[Weyuker 82] E. Weyuker. *On Testing Non-testable Programs*. The Computer Journal, vol. 25, no. 4, 1982.

[Williams 05] N. Williams, B. Marre, P. Mouy & M. Roger. *PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis*. In Proc. Dependable Computing - EDCC'05, 2005.

[Wotawa 10] F. Wotawa, M. Nica & B.K. Aichernig. *Generating Distinguishing Tests Using the Minion Constraint Solver*. In Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, pages 325–330, april 2010.

[Yates 89] D. Yates & N. Malevris. *Reducing The Effects Of Infeasible Paths In Branch Testing*. In Proc. of Symposium on Software Testing, Analysis, and Verification (TAV3), volume 14(8) of *Software Engineering Notes*, pages 48–54, Key West, FL, Dec. 1989.