Université de Franche-Comté **UFR Sciences et Techniques**

École doctorale SPIM

Classification de menaces d'erreurs par analyse statique, simplification syntaxique et test structurel de programmes

THÈSE

présentée et soutenue publiquement le 13 décembre 2011

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Omar Chebaro

Composition du jury

Président :	Catherine Dubois, Professeur à l'ENSIIE
Directeurs :	Jacques Julliand, Professeur à l'Université de Franche-Comté Alain Giorgetti, Maître de conférences à l'Université de Franche-Comté Nikolai Kosmatov, Ingénieur-chercheur au CEA Saclay
Rapporteurs :	Pascale Le Gall, Professeur à l'Université d'Evry-Val d'Essonne Marie-Laure Potet, Professeur à l' ENSIMAG
Examinateur :	Fatiha Zaïdi, Maître de conférences à l'Université Paris-Sud XI

Laboratoire d'Informatique de l'Université de Franche-Comté — CNRS FRE 2661

Mis en page avec la classe thloria.

Remerciements

Je tiens à remercier les membres de mon jury de thèse : Catherine Dubois qui m'a fait l'honneur de présider ce jury, Pascale Le Gall et Marie-Laure Potet qui ont accepté le rôle de rapporteur, et Fatiha Zaidi qui a bien voulu examiner mon travail.

Ma gratitude profonde et sincère à mes directeurs de thèse M. Jacques Julliand et M. Alain Giorgetti pour m'avoir consacré avec patience leur précieux temps et leurs conseils encourageants et pour leur implication dans le suivi régulier de mes travaux.

J'exprime ma gratitude à M. Nikolai Kosmatov, mon encadrant au CEA. Ses orientations, ses conseils et sa disponibilité ont contribué au bon déroulement de cette thèse.

Je tiens à remercier tous ceux qui ont permis à ces travaux d'aboutir, par leurs conseils, leurs contributions et leurs encouragements :

- Merci à tous les membres du laboratoire de sûreté de logiciels du CEA LIST pour leur accueil et pour avoir toujours été disponibles, notamment Patrick Baudin, Bernard Botella, Loïc Correnson, Anne Pacalet, Pascal Cuoq, Bruno Marre, Benjamin Monate, Virgile Prevosto, Muriel Roger, Julien Signoles, Nicky Williams.

- Merci à tous les compagnons de course à pied, activité sans laquelle je n'aurais pu tenir, notamment Bernard, Mickaël, Nikolai, Patricia, Paolo, Richard et Virgile.

Je ne peux pas omettre d'exprimer également un cordial remerciement à toutes les personnes qui m'ont accordé leur aide de près ou de loin.

Enfin, je garde une place toute particulière pour ma famille et mes proches, pour tout ce qu'ils ont pu (et continuent à) m'apporter. Merci à mes sœurs pour leur grand soutien et leurs encouragements. Enfin, merci à mes parents pour leurs prières et pour m'avoir offert toutes les bonnes conditions pour réussir mes études et pour m'avoir apporté, en plus de leur amour, le goût du travail.

À ma famille ...
À tous ceux qui me sont chers.

Table des matières

Table des figures

Première partie

Contexte et problématique

Chapitre 1

Introduction

Sommaire

Cette thèse est l'aboutissement de recherches sur la validation des logiciels, qui est une partie cruciale et encore mal résolue dans le cycle de leur développement. L'automatisation des tâches de validation vise à réduire le temps consacré à cette étape et à améliorer sa qualité. La notion de vérification date des années soixante-dix avec les travaux de Dijkstra [Dij75], Floyd [Flo63] et Hoare [Hoa69]. Celle-ci s'est étalée sur plus de trois décennies de découvertes qui a vu naître un nouveau thème de la science informatique appelée "vérification formelle". Deux grandes classes de techniques de vérification et de validation se sont démarquées au cours de ces dernières années : l'analyse statique et l'analyse dynamique. Ces deux grandes classes de techniques ont longtemps été considérées et étudiées comme deux domaines séparés.

L'analyse statique est généralement utilisée pour prouver la correction d'un programme. Elle examine le code du programme et raisonne sur tous les comportements possibles qui pourraient survenir lors de l'exécution. La vérification de programmes étant en général indécidable, il est souvent nécessaire d'utiliser des sur-approximations. L'analyse statique est prudente et sûre : les résultats peuvent être moins précis que ce que l'on souhaite,

mais ils sont garantis pour toutes les exécutions. Par exemple, la vérification statique d'une propriété par preuve apporte dans le cas général deux sortes de réponses, soit que la propriété est satisfaite, soit qu'elle ne sait pas. Dans le second cas, elle ne sait pas dire si la propriété est fausse ou si elle n'est pas parvenue à la prouver.

L'analyse dynamique révèle la présence d'erreurs. Elle fonctionne en exécutant un programme et en observant certaines de ses exécutions. Elle est en général incomplète en raison du grand nombre de cas de tests possibles. L'analyse dynamique est précise sur la partie analysée, mais elle effectue des sous-approximations en n'explorant qu'un sous-ensemble des exécutions. L'analyse dynamique examine le comportement exact d'exécution du programme pour le cas de test correspondant, mais n'en examine que quelques-uns. Les deux méthodes, statique et dynamique, utilisent des abstractions, mais l'analyse statique essaie de construire des abstractions (sur-approximations) préservant la propriété vérifiée alors que l'analyse dynamique utilise des abstractions (sous-approximations) non conservatives.

Les points forts des deux méthodes sont complémentaires. L'analyse statique [NNH99] est imprécise mais correcte. En effet, ses résultats peuvent être plus faibles que souhaités, mais ils présentent l'avantage d'être valables pour l'ensemble des exécutions. L'analyse dynamique quant à elle est une méthode précise mais incomplète. En effet, pour un cas de test donné, l'analyse dynamique peut examiner le comportement exact du programme. Cependant cette analyse est incomplète en raison de l'analyse de quelques-unes des exécutions parmi le grand nombre (voire une infinité) de cas possibles.

Pour résumer, si aucune menace d'erreur (d'un type particulier) n'est levée lors de l'analyse statique dans aucun des comportements possibles d'un programme donné, alors il est certain que ce programme ne contient pas ce genre d'erreurs. Cependant, les erreurs potentielles signalées par cette analyse peuvent s'avérer de fausses alarmes. Par contre, même si l'analyse dynamique ne peut en général pas prouver l'absence d'erreur dans un programme, une erreur détectée par cette analyse est forcément réelle.

Dans le débogage manuel, les développeurs de logiciel expérimentés tentent généralement de localiser et d'isoler une erreur en simplifiant le programme autant que possible [Wei82], par exemple en commentant les instructions non pertinentes avant et après l'instruction suspectée. Ensuite, on rejoue le même test ou le même scénario pour voir si les comportements erronés sont conservés. La raison de l'erreur peut être beaucoup plus facile à comprendre lorsqu'on analyse une version réduite du programme. Cette démarche manuelle pourrait être automatisée à l'aide de la simplification syntaxique de programme (*program slicing*). La simplification syntaxique est une technique de transformation de programmes qui permet de simplifier un programme afin d'isoler un comportement spécifique.

Notre objectif dans cette thèse est de fournir une technique automatisée pour détecter les erreurs de division par zéro et les accès hors limites de tableaux à l'exécution dans les programmes C ou prouver leur absence. Nous avons pensé pouvoir atteindre cet objectif en combinant une analyse statique, une analyse dynamique et des simplifications de pro-

gramme.

Nous présentons dans cette thèse une combinaison originale qui permet de rendre la recherche d'erreurs automatique, plus précise et plus efficace en temps que la recherche avec des techniques d'analyse statique ou d'analyse dynamique prises séparément.

Un programme donné contient un certain nombre de menaces d'erreurs. Une menace est dite **classée** par une analyse si elle est confirmée ou infirmée. L'analyse étant imprécise ou incomplète, elle peut laisser un certain nombre de menaces avec le statut de menace sans pouvoir ni les confirmer, ni les infirmer. Nous dirons qu'un diagnostic d sur un programme est **plus précis** qu'un diagnostic d' si le nombre de menaces d'erreur classées dans d est plus grand que celui de d'.

Ce chapitre détaille le contexte et la problématique de cette thèse, précise nos motivations et nos contributions, et enfin présente le plan de ce mémoire.

1.1 Contexte

Les systèmes informatisés sont de plus en plus présents autour de nous. Ils se sont imposés dans notre quotidien à tel point qu'ils se sont rendus indispensables. En cas de dysfonctionnement, divers coûts et dommages peuvent arriver. De plus certains systèmes critiques mettent en jeu des vies humaines (systèmes de transport, systèmes médicaux, centrales nucléaires, ...) ou d'importantes sommes d'argent (navettes spatiales, transactions bancaires, ...). Pour illustrer les conséquenses financières et humaines d'une erreur logicielle, nous présentons trois exemples que nous avons emprunté à la thèse d'Olivier BOUISSOU [Bou08]. Un exemple bien connu est l'explosion d'Ariane 5 le 4 juin 1996. Ce projet a duré dix années et a coûté sept milliards de dollars. L'échec du lancement inaugural d'Ariane 5 a été causé par une erreur de dépassement de capacité dans le module responsable du calcul du mouvement de la fusée.

Un autre échec logiciel connu est celui du missile Patriot qui était utilisé par l'armée américaine pour se protéger des missiles irakiens durant la guerre du Golfe. Le 25 février 1991, un missile Patriot a échoué dans l'interception d'un missile irakien, et a par conséquent provoqué la mort de 28 soldats américains. La cause de l'échec identifiée était une perte de précision lors de la conversion d'un entier vers un nombre flottant codé sur 24 bits.

Un autre échec logiciel connu est le blocage du croiseur lance-missiles de l'armée américaine "USS Yorktown". Ce croiseur est équipé d'un système de contrôle et de maintenance automatique. En septembre 1997, le système informatique s'est bloqué, en raison d'une division par zéro, entraînant l'immobilisation du navire pendant plusieurs heures.

La division par zéro est un exemple classique d'erreur à l'exécution (*Run Time Error*, ou RTE) qui bloque complètement un programme. Une autre erreur classique est l'erreur

de segmentation qui apparaît quand on essaie d'accéder à une zone mémoire indisponible. De même, ce type d'erreur peut bloquer complètement le programme. Les erreurs à l'exécution comme la division par zéro ou les erreurs de segmentation sont donc très problématiques et les détecter ou prouver leur absence est un enjeu capital, d'autant plus que c'est souvent en exploitant ce genre d'erreurs que des failles de sécurité sont trouvées.

Les travaux présentés dans cette thèse entrent dans le cadre de la vérification des programmes C. Ils permettent de classer les menaces d'erreurs par analyse statique, simplification syntaxique et test structurel. Nous présentons tout d'abord quelques caractéristiques du langage C qui induisent des problèmes de sécurité. Puis nous présentons brièvement quelques techniques de validation et de vérification.

1.1.1 Langage C

Le langage C est un langage de programmation impératif. Il a été introduit dans les années soixante-dix [KR78]. La première norme pour le langage C a été publiée par l'ANSI (*American National Standards Institute*) en 1989. Cette norme est appelée ANSI C ou C89 [Ame89]. Elle a été adoptée un an plus tard par l'organisation internationale de normalisation (ISO), donc elle est aussi connue comme ISO C ou C90 [ISO90]. La norme actuelle a été adoptée en 1999 par l'ISO et est connue sous l'acronyme C99 [ISO99].

Le langage C était initialement conçu pour le développement des systèmes d'exploitation. Depuis, C est devenu un des principaux langage de programmation pour le développement des logiciels, d'autant plus que la plupart des langages informatiques modernes comme PERL, JAVA ou OCAML s'appuient dans leur environnement d'exécution sur une couche de programmes C.

C est qualifié de langage bas niveau, car le programmeur peut manipuler directement la mémoire. Un programme C peut être converti directement en langage machine (binaire) grâce à un **compilateur**.

Bien que C soit un langage ancien maintenant, et que beaucoup de puissants langages informatiques spécialisés aient été conçus depuis, la connaissance de C était toujours la deuxième compétence la plus demandée dans les offres d'emploi des programmeurs en 2007 [Ent07], et c'est le langage de programmation à usage général le plus demandé. C reste toujours un langage très utilisé, avec des applications dans de nombreux secteurs différents, dont certains dépendent de façon critique des logiciels.

Avec le nombre élevé de programmes C critiques dans l'industrie, les techniques et outils d'analyse s'intéressent de plus en plus à ce langage. À cause de son orientation bas niveau, le langage C ne protège pas contre les erreurs à l'exécution et les défauts d'accès mémoire, qui sont la source principale d'erreurs. Le langage C est un langage faiblement typé qui manque de mécanismes de sécurité. Nous détaillons cet aspect ci-dessous.

Manque de mécanismes de sécurité

Le langage C a été conçu pour permettre un accès bas niveau aux structures de données, permettant d'accéder à chaque octet de la mémoire. Par conséquent, la gestion de mémoire est laissée à la charge du programmeur et pour cette raison, il était impossible d'appliquer un typage fort.

En raison du manque de support pour la gestion de la mémoire en C, il reste entièrement sous la responsabilité du programmeur d'accéder uniquement à la mémoire valide, c'est-à-dire à la mémoire correctement allouée et non encore libérée. De plus, il n'existe aucun mécanisme pour connaître la taille d'un tableau durant l'exécution, ce qui provoque fréquemment des erreurs de type *buffer overflow* dans les programmes C, où un *buffer* fait l'objet d'un accès au-delà de ses limites.

L'absence de typage fort en C permet d'accéder à une variable de type t via une autre variable de type t' incompatible avec t. Le standard C [ISO99] énumère de nombreux exemples de violations de types de données, dont la plus simple est la lecture des données non initialisées. Un programme peut lire quelques bits et les interpréter comme une valeur de type t. Cette faille pourrait être révélée au contrôle de type, par exemple, si le compilateur rejetait certains modèles de bits comme des valeurs invalides de type t. Mais C accepte ces confusions et risque donc d'accepter des valeurs erronées qui peuvent déclencher une erreur ou donner de mauvais résultats.

1.1.2 Techniques de validation et de vérification

Beaucoup de techniques de validation et de vérification sont actuellement utilisées. Les principales sont l'analyse dynamique par génération et exécution de tests, le *model-checking* de propriétés, l'interprétation abstraite et la preuve interactive. Cette partie sera approfondie dans le chapitre 2.

Analyse dynamique

L'analyse dynamique consiste à soumettre un programme à un ensemble d'exécutions pour un ensemble de valeurs des variables d'entrées pour vérifier qu'il se comporte sur ces entrées comme le décrit sa spécification. C'est la technique la plus utilisée dans l'industrie pour la validation de programmes. Elle est en général incomplète dû au grand nombre (voire une infinité) de cas de test possibles. Aujourd'hui, des normes de sûreté de systèmes critiques à base de composants électroniques demandent des critères de confiance supérieurs à ceux fournis par l'analyse dynamique [Fal01].

Model-checking

Le model-checking [QS82, CE82] d'un système est une technique permettant de vérifier si un modèle satisfait sa spécification. Un modèle est l'ensemble des états possibles du système, et des transitions entre états qui décrivent ses évolutions avec les propriétés que vérifie chaque état. Le model-checking couvre l'espace d'états du système et ses transitions afin d'analyser toutes les exécutions possibles du système. De plus, mise à part la modélisation du système et l'expression de sa spécification, le model-checking est une technique entièrement automatique pour les systèmes à nombre fini d'états, ce qui la rend très utilisée pour certaines classes d'applications industrielles aujourd'hui [HJM+02, BHJM07]. Cependant la taille de l'espace d'états d'un système formé de plusieurs composants est la multiplication des tailles des espaces d'états de chacun. Un phénomène d'explosion combinatoire apparaît alors sur les grands systèmes. Le model-checking de tels systèmes peut alors souffrir d'une (trop) grande complexité en temps ou en espace.

Interprétation abstraite et abstraction à partir de prédicats

L'interprétation abstraite calcule en temps fini un sur-ensemble des comportements d'un programme, alors que le problème du calcul exact des comportements est indécidable [Ric53]. Elle s'appuie sur les théories du point fixe et des domaines pour introduire des approximations. L'interprétation abstraite est aujourd'hui utilisée avec succès dans l'industrie, mais souffre de taux élevé de fausses alarmes. L'abstraction est effectuée à partir de prédicats atomiques définissant des abstractions des domaines des variables. L'abstraction à partir de prédicats consiste à abstraire à partir de prédicats définis par composition de prédicats atomiques.

Preuve interactive

On annote le code source des programmes en y insérant des hypothèses et des conclusions dans une logique à la Hoare [Hoa69]. Un calcul de plus faible précondition établit alors des formules de la logique des prédicats qui doivent être vérifiées afin de prouver la correction des programmes. La preuve interactive permet de prouver une large classe de propriétés sur les programmes mais requiert un utilisateur expert des méthodes de preuve pour annoter les programmes. L'automatisation n'est que partielle. En particulier, l'utilisateur doit en général écrire les invariants d'itération.

Combinaisons de méthodes

Récemment, plusieurs outils ont proposé des méthodes pour la vérification de programmes. Les outils les plus connus sont : BLAST [BHJM07] qui combine l'abstraction à partir des prédicats et le model-checking et YOGI [GHK+06, NRTT09] qui combine l'abstraction à partir des prédicats et la génération de tests. Avec ces outils, la non atteignabilité des états d'erreur est spécifiée par une propriété de sûreté et les outils ne

vérifient qu'une propriété à la fois.

[CS05] présente une nouvelle combinaison d'analyse statique et de génération de tests pour la détection des erreurs à l'exécution. Cet outil n'est ni correct ni complet. Les utilisateurs doivent faire face à un travail lourd d'annotation et un taux élevé de fausses alarmes.

C'est dans cette catégorie d'approches que se situe cette thèse, en fournissant une nouvelle combinaison d'analyse de valeurs, de simplification syntaxique et de test structurel pour la vérification des programmes écrits en C.

1.2 Problématiques et motivations

L'équipe d'enquête chargée de déterminer les causes de l'explosion d'Ariane 5 a affirmé dans sa conclusion que les tests effectués en cours du développement d'Ariane 5 ne comprenaient ni une analyse adéquate ni des tests du système de contrôle de vol complet, ce qui aurait peut-être suffi pour détecter la défaillance potentielle [LLF+96].

Dans le cas d'Ariane 5, une erreur du programme a eu des conséquences financières très fortes. Dans le cas du missile Patriot, les pertes humaines étaient catastrophiques. Il est donc primordial, avant d'utiliser des programmes dans des systèmes embarqués, de s'assurer de leur bon fonctionnement. Le génie logiciel préconise le recours aux méthodes formelles permettant de vérifier les systèmes pour assister les ingénieurs de développement.

Etant incomplet, le processus de validation par des tests n'est pas suffisant pour évaluer la fiabilité des systèmes logiciels, surtout avec l'augmentation de la taille des logiciels. Ce processus ne peut pas être fondé sur des techniques d'abstraction trop imprécises. La tendance actuelle des industriels est de se concentrer sur les techniques correctes. Ces techniques devraient pouvoir passer à l'échelle pour des programmes de grande taille. Elles devraient aussi s'appliquer à des programmes C existants de façon automatique. Les utilisateurs ne devraient pas avoir besoin d'intervenir pour guider la vérification et pour produire des résultats précis.

Une dernière motivation importante de ce travail est de fournir automatiquement à l'ingénieur de validation non seulement les données de test et le chemin du programme menant à une erreur, mais aussi autant d'informations que possible sur l'erreur détectée. Par exemple, l'erreur peut être illustrée sur un programme plus simple, avec un chemin plus court conduisant à l'instruction erronée et un ensemble de contraintes réduit portant sur les variables utiles seulement. La plupart des outils de vérification modernes ne font pas ces simplifications et ne fournissent pas ces informations simplifiées, qui peuvent réduire considérablement le temps nécessaire à un développeur de logiciels pour analyser et corriger l'erreur. Dans notre méthode, nous proposons d'introduire une phase de simplification syntaxique, appelée plus couramment *slicing*, afin de simplifier le programme analysé et les contre-exemples produits. Le programme simplifié est appelé *slice*. L'apport

de la simplification est particulièrement intéressant dans le cas de la génération automatique de code à partir de modèles, où le développeur qui investigue la source d'erreur n'a pas de connaissance profonde du code source analysé.

1.3 Contributions

Nous traitons les problématiques discutées ci-dessus en mettant en place une méthode automatique combinant la complétude de l'analyse de valeurs, la simplification par transformation syntaxique et la précision de l'analyse dynamique.

Notre première contribution est une combinaison originale des deux techniques d'analyse statique et d'analyse dynamique afin d'améliorer la précision en classant plus de menaces. Notre deuxième contribution est l'intégration de la simplification syntaxique (*slicing*) dans notre combinaison. La troisième contribution de cette thèse est la formalisation de la méthode proposée et la preuve de sa correction. La quatrième contribution consiste en l'implémentation de la méthode dans l'outil nommé SANTE. Notre dernière contribution consiste à évaluer en pratique par des expérimentations les techniques proposées dans cette thèse en utilisant l'implémentation.

1.3.1 Combinaison d'analyse de valeurs et d'analyse dynamique

Dans cette combinaison, l'analyse statique (analyse de valeurs) signale les instructions risquant de provoquer des erreurs à l'exécution par des alarmes, dont certaines peuvent être de fausses alarmes. Puis l'analyse dynamique (génération de tests) est utilisée pour confirmer ou rejeter ces alarmes. Cette contribution a été publiée dans [CKGJ10b], où nous avons présenté SANTE et la technique de génération de tests guidée par les alarmes, ainsi que les premières expérimentations.

1.3.2 Intégration de la simplification syntaxique

Appliquée à des programmes de grande taille, la génération de tests peut manquer de temps ou d'espace mémoire avant de confirmer certaines alarmes comme de vraies erreurs ou de conclure qu'aucun chemin faisable ne peut atteindre l'état d'erreur de certaines alarmes et donc rejeter ces alarmes. Pour surmonter ce problème, nous proposons de réduire la taille du code source par le *slicing* avant de lancer la génération de tests. Le *slicing* transforme un programme en un autre programme plus simple, qui est équivalent au programme initial par rapport à certains critères. Les critères utilisés dans notre méthode préservent une ou plusieurs instructions menaçantes.

Quatre utilisations du *slicing* sont étudiées. La première utilisation est nommée *all*. Elle consiste à appliquer le *slicing* une seule fois, le critère de simplification étant l'en-

semble de toutes les alarmes du programme qui ont été détectées par l'analyse statique. L'inconvénient de cette utilisation est que la génération de tests peut manquer de temps ou d'espace et les alarmes les plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes. Dans la deuxième utilisation, nommée *each*, le *slicing* est effectué séparément par rapport à chaque alarme. Cependant, la génération de test est exécutée pour autant de programmes que d'alarmes et il y a un risque de redondance d'analyse si des alarmes sont incluses dans plusieurs programmes. Ces contributions ont été publiées dans [CKGJ11], où nous avons présenté la méthode intégrant le *slicing* avec les utilisations *all* et *each*.

Pour pallier les inconvénients des deux options présentées, nous avons étudié les dépendances entre les alarmes et nous avons introduit deux utilisations avancées du *slicing*, nommées *min* et *smart*, qui exploitent ces dépendances. Dans l'utilisation *min*, le *slicing* est effectué par rapport à un ensemble minimal de sous-ensembles d'alarmes. Ces sous-ensembles sont choisis en fonction de dépendances entre les alarmes et l'union de ces sous-ensembles couvre l'ensemble de toutes les alarmes. Avec cette utilisation, on a moins de *slices* qu'avec *each*, et des *slices* plus simples qu'avec *all*. Cependant, l'analyse dynamique de certaines *slices* peut encore manquer de temps ou d'espace avant de classer certaines alarmes, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple permettrait de les classer. L'utilisation *smart* consiste à appliquer l'utilisation précédente itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire. Lorsqu'une alarme ne peut pas être classée par l'analyse dynamique d'une *slice*, des *slices* plus simples sont calculées. Ces deux options sont détaillées dans [CKGJ12].

1.3.3 Formalisation

Nous formalisons la méthode proposée et nous prouvons sa correction. Nous prouvons que lorsqu'une alarme est classée sans risque sur un programme simplifié, ce classement serait le même sur le programme initial. Lorsqu'une erreur est détectée pour une alarme sur le programme simplifié, une simple confirmation par exécution du programme initial permet de vérifier le statut de cette alarme dans le programme initial.

1.3.4 Implémentations

Ces travaux sont implantés dans SANTE, nouvel outil qui relie l'outil de génération de test PATHCRAWLER [WMM03, WMMR05, BDHTH+09] et la plate-forme d'analyse statique FRAMA-C [Fra11, CS09]. Cette contribution a été publiée dans [Che10], où nous avons présenté l'outil SANTE CONTROLLER.

1.3.5 Évaluation expérimentale

Les expérimentations ont été menées en utilisant l'implémentation pour valider notre approche sur un ensemble de programmes et mettre en évidence ses limites. Nous comparons notre méthode avec l'analyse de valeurs seule et l'analyse dynamique seule sur des critères tels que le nombre de menaces restantes non classées et le temps requis par chaque analyse. Nous mesurons aussi l'effet de la simplification syntaxique sur la longueur des chemins analysés et leur nombre, ainsi que sur la taille du programme. Ces résultats sont présentés dans [CKGJ10b] et [CKGJ12].

1.4 Plan de la thèse

Cette thèse est structurée en deux parties : la première est consacrée au contexte et à l'état de l'art et comporte trois chapitres :

- Le présent chapitre, **chapitre 1** introduit le contexte scientifique et les problématiques abordées lors de cette thèse. Il énonce les contributions et les principaux résultats.

- Le **chapitre 2** présente les différentes techniques de vérification et les travaux existants connexes aux travaux présentés dans cette thèse.

- Le **chapitre 3** détaille le fonctionnement de l'outil PathCrawler de génération de tests structurels, et celui de la plate-forme Frama-C pour l'analyse statique des programmes C.

La **deuxième partie** est consacrée aux contributions apportées par cette thèse. Elle est composée de quatre chapitres :

- Le **chapitre 4** présente de manière détaillée et formalisée la méthode sante qui combine l'analyse de valeurs, le *slicing* et la génération de tests structurels pour la vérification des programmes C. Dans le cadre de cette contribution, nous proposons quatre usages du *slicing*, étudions les dépendances entre les alarmes et démontrons la correction de la méthode.

- Le **chapitre 5** décrit l'implémentation des méthodes proposées dans cette thèse. Tout d'abord, nous présentons le travail effectué pour rapprocher les deux outils, les difficultés d'intégration et les solutions adoptées. Nous présentons ensuite les adaptations techniques visant à traiter les cas d'erreurs dans PathCrawler. Nous présentons enfin notre implémentation sante controller qui contrôle les différentes analyses et fournit les diagnostics.

- Le **chapitre 6** présente les résultats de l'utilisation de la méthode sante sur des

exemples de programmes existants. Nous analysons les résultats des expérimentations afin de faire un bilan de l'apport et des limites des méthodes proposées dans ce mémoire.

– Le **chapitre 7** dresse un bilan de ce travail et en suggère diverses perspectives.

L'annexe A comporte des extraits simplifiés du code source du greffon SANTE CONTROLLER qui implémente la méthode SANTE.

Chapitre 2

État de l'art

Sommaire

2.1 Introduction

Depuis le début de l'ingénierie logicielle, différentes méthodes d'analyse de programmes et de modèles ont été développées pour les vérifier et les valider. On peut les classer en deux catégories, les méthodes d'analyse statique d'une part et les méthodes d'analyse dynamique d'autre part. Dans chacune des catégories on trouve un ensemble de techniques

pour effectuer ces analyses.

Pour faire de l'analyse statique, on dispose des techniques suivantes.

1. L'interprétation abstraite [CC77, CC92] est utilisée par exemple pour faire de l'analyse de valeurs des variables d'un programme ou d'un modèle en approximant leurs domaines. Cette analyse permet de détecter certaines erreurs comme les divisions par zéro, les accès en-dehors des bornes des tableaux, des dépassements de capacité des implantations des types de données, etc.

2. Le calcul de plus faible précondition [Dij75, Hoa69] est utilisé pour prouver des programmes relativement à des spécifications pré-post ou pour calculer des abstractions de modèles.

3. Le model-checking [QS82, CE82] permet de vérifier algorithmiquement des propriétés de sûreté, de vivacité, d'équité, etc.

4. L'abstraction à partir de prédicats [GS97] est utilisée pour calculer des abstractions de modèles à nombre d'états infini et permettre ainsi de vérifier par model-checking des classes de propriétés qu'elle préserve.

5. Le calcul de flot de contrôle et de données de programmes permet d'effectuer des vérifications de propriétés existentielles. Il permet également des simplifications syntaxiques, appelées *slicing* [Wei81, Wei82], relativement à des critères de simplification. Les programmes ainsi simplifiés peuvent être vérifiés plus facilement. Noter que nous utilisons une technique de *slicing* statique uniquement, mais celui-ci peut également être dynamique [KL88].

L'analyse dynamique est basée sur des techniques d'exécution du programme ou de simulation [WB89] d'un modèle. Le test de programmes [Bei90] est une technique dynamique basée sur la sélection de quelques chemins d'exécution concrets. L'utilisation de l'exécution symbolique pour le test des programmes a été introduite en 1976 par [Cla76] et [Kin76]. En général, le nombre de chemins étant très grand, voire infini, [WMMR05, GKS05] ont proposé des techniques d'exécution symbolique qui permettent de faire une sorte d'analyse structurelle des programmes en sélectionnant une exécution par chemin du flot de contrôle d'un programme. Le graphe peut être couvert totalement en l'absence d'itération non bornée. Dans le cas contraire, l'exécution symbolique examine partiellement l'ensemble infini de chemins du graphe de flot de contrôle.

Les méthodes statiques et les méthodes dynamiques ont des avantages et des inconvénients différents. L'idée de les combiner pour associer leurs avantages et combattre leurs inconvénients est une voie de recherche active et fructueuse dans le domaine de l'analyse de programmes. Par exemple, l'exécution symbolique qui met en jeu des résolutions de contraintes peut échouer parce qu'elle ne dispose pas de la bonne théorie et de la procédure

de décision adaptée. Il est alors possible d'utiliser l'exécution dynamique pour simplifier le système de contraintes et permettre sa résolution.

Dans ce chapitre d'état de l'art, dans la partie 2.2, nous avons choisi de présenter brièvement les techniques statiques d'interprétation abstraite et de preuve de programmes. Dans la partie 2.3, nous présentons les techniques d'analyse dynamique de génération de tests structurels et fonctionnels. Dans la partie 2.4, nous présentons la technique de simplification syntaxique appelée *slicing*. Enfin, dans la partie 2.5, nous décrivons des travaux combinant des techniques d'analyse statique et dynamique et les outils automatiques de vérification et de validation de programmes qui en sont issus.

2.2 Analyse statique

L'analyse statique permet de trouver statiquement (c'est-à-dire sans exécuter) des propriétés devant être vraies pour toutes les exécutions du programme. Nous présentons ici trois techniques de vérification qui utilisent l'analyse statique. Il s'agit de l'interprétation abstraite (section 2.2.1), de l'abstraction à partir de prédicats (section 2.2.2) et de la preuve (section 2.2.3).

Les propriétés à vérifier peuvent être de différentes natures. Les plus simples sont sûrement les propriétés de sûreté (*safety*), où l'on cherche des invariants sur les valeurs des variables du programme, c'est-à-dire une plage de valeurs X telle qu'au cours de toutes les exécutions du programme, la variable x reste dans X. Un tel invariant permet par exemple de montrer qu'il n'y a pas de risque d'erreur à l'exécution (*runtime error* ou RTE) dû à une division par zéro : si $0 \notin X$, alors la variable x ne pourra pas provoquer une division par zéro.

Un autre type de propriétés de sûreté est l'absence de RTE due à un débordement de buffer (*buffer overflow*). Cela peut arriver, par exemple, quand on essaie d'écrire le $n^{\text{ième}}$ élément dans un tableau qui contient m cases avec $m < n$, ou quand on essaie de copier une chaîne de caractères de longueur n dans une autre chaîne de longueur $m < n$.

Un autre type de propriétés de sûreté est lié au comportement des nombres flottants. En fait, les nombres flottants sont censés être une représentation des nombres réels mais sur un nombre fini de bits. Cela introduit naturellement de l'imprécision (ou une déviation), et les calculs effectués dans l'arithmétique des nombres flottants diffèrent, parfois assez sensiblement, des calculs effectués dans l'arithmétique réelle [IA85]. Mesurer cette erreur peut permettre de corriger certaines erreurs de conception et/ou d'implémentation d'un algorithme. Les propriétés de sûreté permettent de prouver qu'aucun mauvais comportement ne va arriver lors de l'exécution du programme.

Outre les propriétés de sûreté, il y a les propriétés de vivacité (*liveness*) [CPR06b], qui peuvent aussi être intéressantes à prouver. Ainsi, la terminaison du programme est souvent

une question importante [CPR06a], de même que la détection de code mort (c'est-à-dire des morceaux du code qui ne seront jamais exécutés).

Les problèmes liés aux propriétés mentionnées ci-dessus (quelle est l'erreur maximale de calcul ? Y a-t-il une division par zéro ?, ...) sont indécidables [Lan92] dans le cas général. En d'autres mots, on ne peut pas trouver d'algorithme capable de répondre correctement pour tout programme P à la question suivante : y a-t-il une exécution de P qui mène à une division par zéro ?

2.2.1 Interprétation abstraite

Pour contourner le problème d'indécidabilité, la théorie de l'interprétation abstraite [CC77, CC92] permet de construire une méthode qui, à la même question, répondra OUI, NON ou PEUT-ETRE. Si la méthode répond OUI, alors il y aura effectivement une division par zéro lors d'une des exécutions du programme. Si la méthode répond NON, alors le programme ne contient sûrement pas une division par zéro. Enfin, si la méthode répond PEUT-ETRE, c'est qu'on n'a pu prouver ni l'un ni l'autre des deux premiers cas. C'est ce qu'on appelle une **alarme** : il est possible qu'une des exécutions du programme produise une division par zéro, mais nous n'avons été capable ni de le confirmer ni de l'infirmer. L'erreur signalée par une alarme peut ne jamais apparaître à l'exécution, dans ce cas on l'appelle **fausse alarme**.

On ne calcule donc pas la propriété exacte mais une abstraction de cette propriété, en imposant la contrainte de sûreté suivante : *"la propriété abstraite calculée ne doit oublier aucune exécution concrète."*. Ainsi, si la réponse de la méthode est NON pour le programme P, on doit alors pouvoir affirmer qu'aucune exécution de P ne provoque de division par zéro. L'interprétation abstraite appliquée à l'analyse de valeurs consiste à calculer à chaque ligne du code une sur-approximation de l'ensemble des valeurs prises par une variable x en cette ligne lors de toutes les exécutions du programme.

Il existe de nombreux outils d'analyse statique par interprétation abstraite. La majorité d'entre eux sont restés au stade de prototypage servant à des fins de recherche dans les laboratoires universitaires. Cependant, quelques uns d'entre eux ont pu être utilisé dans le cadre de projets industriels de grande taille. Nous décrivons ici trois. Chacun de ces outils traite un type particulier de propriété.

PolySpace C Verifier

PolySpace C Verifier [Pol11] est un outil commercial initialement conçu pour détecter les erreurs à l'exécution dans les logiciels embarqués. Il est maintenant intégré à Matlab/-Simulink qui prouve l'absence d'erreurs à l'exécution. Hormis le fait que l'interprétation abstraite est utilisée et que les algorithmes sont basés sur les travaux de Patrick Cousot [CC77, CC92], peu d'autres détails sur l'algorithme sont fournis.

PolySpace C Verifier est correct (*sound*), il ne laisse pas d'erreurs non signalées, mais signale beaucoup de fausses alarmes (en moyenne une fausse alarme pour toutes les 12 lignes de code d'après [ZLL04]). Il peut tourner pendant des heures, voire des jours, sur des programmes de taille moyenne (quelques milliers de lignes de code). Il gère la plupart des constructions C ainsi que Ada et C++, et effectue une analyse d'alias [Pol11, ZLL04].

ASTREE

L'outil ASTREE [AST11, CCF+05] (pour Analyseur statique de logiciels temps-réel embarqués) est un analyseur statique, développé à l'Ecole Normale Supérieure. Il prouve l'absence d'erreur à l'exécution pour des programmes embarqués temps-réel écrits en C. ASTREE peut détecter des erreurs spécifiques au langage C (division par zéro, dépassement de capacité pour un tableau), des erreurs liées à la représentation en machine des données (dépassement par exemple de la plus grande valeur représentable) ou encore le non-respect de propriétés définies par l'utilisateur sous forme d'assertions. ASTREE ne vise pas à être un analyseur générique mais au contraire, il s'est spécialisé pour traiter une classe précise de programmes pour lesquels une RTE est critique. De par cette spécialisation et grâce à un choix de domaines abstraits spécifiques à l'analyse des logiciels visés (voir par exemple [Fer04] pour les filtres intervenant dans les logiciels de contrôle-commande), ASTREE atteint une très grande précision d'analyse, ce qui permet son application pour des projets industriels de très grande taille [CCF+09].

Fluctuat

L'outil Fluctuat [LIS11, BGP+09, GP11] est développé au laboratoire de modélisation et analyse des systèmes en intéraction du CEA LIST. Il mesure l'erreur due à l'utilisation des nombres flottants au lieu des nombres réels lors de l'exécution d'un programme écrit en C. L'erreur est même décomposée en une "série d'erreurs" qui indique la contribution de chaque instruction du programme à l'erreur finale. On peut ainsi visualiser directement quelle instruction cause la plus grande erreur ce qui permet un déboguage précis du programme en cas d'erreur finale complexe. L'utilisation de domaines très spécialisés pour l'analyse des erreurs de calcul et un grand choix de paramètres concernant l'analyse permettent d'obtenir des résultats précis pour une grande gamme de programmes [GP06].

2.2.2 Abstraction à partir de prédicats

L'abstraction à partir de prédicats [GS97, BMMR01] calcule une abstraction par preuve. C'est une autre approche que l'interprétation abstraite. L'interprétation abstraite consiste à abstraire les domaines des variables par des domaines finis et beaucoup plus petits. Par exemple le domaine des entiers pourrait être abstrait par un domaine de trois

valeurs, les entiers négatifs, zéro et les entiers positifs. L'abstraction à partir de prédicats consiste à abstraire l'ensemble d'états du système par un ensemble fini $Pred$ de n prédicats, portant sur l'ensemble Var des variables du programme et non pas sur chaque variable prise séparément. L'ensemble d'états abstraits est fini et borné par 2^n. En général, on n'abstrait pas le compteur de programme *pc*. Un état abstrait d'un programme est alors défini par un $n+1$-uplet composé d'un emplacement de programme *pc* et des n prédicats de $Pred$ ou leur négation. La correspondance entre un état concret c et un état abstrait a est établie par l'évaluation de tous les prédicats dans $Pred$ sur les valeurs concrètes des variables du programme dans l'état c au même emplacement du programme.

Le calcul de la relation de transition abstraite est effectué par preuve de satisfaisabilité de formules établies par calcul de plus faible précondition. Ces preuves sont effectuées en faisant appel aux prouveurs automatiques tels que les solveurs SMT comme Z3 [dMB08] et les prouveurs par déduction tels que Simplify [DNS05].

Prenons l'exemple suivant d'un programme ayant deux variables $Var = \{i, x\}$ et calculons son abstraction à partir de l'ensemble de 2 prédicats $Pred = \{P_1 : i < 0, P_2 : x > 0\}$. À un point donné, ce programme applique l'opération *op* suivante : $x \mathrel{+}= i$.

Considérons un état abstrait $a = (2, (faux, vrai))$ où $pc = 2$ est l'emplacement de programme de l'opération *op*, qui incrémente x par i. Afin de trouver tous les états abstraits successeurs de a, on doit savoir quelles sont les états possibles de P_1 et P_2 parmi les quatre possibilités $((vrai, vrai), (vrai, faux), (faux, vrai), (faux, faux))$ après l'évaluation de l'opération $x \mathrel{+}= i$ pour tous les états concrets représentés par a. Les états concrets représentés par a sont les états concrets qui ont l'emplacement du programme $pc = 2$ et tels que les valeurs des variables satisfont les prédicats $\neg P_1$ $(i \geq 0)$ et P_2 $(x > 0)$.

Comme x est la seule variable modifiée par l'opération *op*, seuls les prédicats qui font référence à x peuvent changer de valeur à la fin de l'opération *op*. Par conséquent, P_1 reste $faux$ dans tous les successeurs de a.

Le prédicat P_2 fait référence à x et les valeurs qu'il peut prendre dans les états successeurs sont donc inconnues. Pour écarter les valeurs qui ne peuvent pas survenir, on construit une formule logique pour chaque valeur possible et on vérifie cette formule à l'aide d'un prouveur. Dans l'exemple, les deux formules correspondant à la valeur $vrai$ et la valeur $faux$ du prédicat P_2 après l'application de *op* à l'état abstrait a sont :

$$\varphi_1 : (i_0 \geq 0) \wedge (x_0 > 0) \wedge (x_1 = x_0 + i_0) \wedge (x_1 > 0)$$

et

$$\varphi_2 : (i_0 \geq 0) \wedge (x_0 > 0) \wedge (x_1 = x_0 + i_0) \wedge (x_1 \leq 0).$$

Dans ces formules, $(i_0 \geq 0) \wedge (x_0 > 0)$ sont les valeurs de $\neg P_1$ et P_2 dans l'état a, c'est-à-dire avant l'évaluation de *op*. $(x_1 = x_0 + i_0)$ est le prédicat avant-après définissant l'opération *op* qui modifie la valeur initiale de x (x_0), et met le résultat dans une nouvelle variable x_1. $(x_1 > 0)$ et $(x_1 \leq 0)$ sont les valeurs potentielles du prédicat P_2 dans les deux états

abstraits successeurs possibles. Dans notre exemple, seule la formule φ_1 est satisfaisable et donc il y a un seul état successeur $(3, (faux, vrai))$, où $pc = 3$ est l'emplacement de programme qui vient juste après l'opération *op*.

Un des outils les plus connus qui utilise l'abstraction à partir de prédicats est SLAM [SLA11, BR02]. Il est présenté dans la section 2.5.1.

2.2.3 Preuve de théorèmes

La preuve de théorèmes utilise des fondements mathématiques pour prouver que les programmes sont conformes à leurs spécifications. Historiquement, l'objectif principal de ces recherches était de garantir la correction des systèmes critiques. Il s'agit d'appliquer successivement des règles de déduction sur une formule écrite dans une logique donnée et permettant au final d'obtenir un verdict sur la vérité d'une formule.

Initialement ces preuves ont été établies à la main en utilisant l'expertise d'un mathématicien ou d'un logicien. Ce processus est partiellement automatisé en utilisant un **prouveur de théorèmes**.

Les approches par preuve de théorèmes vérifient des modèles, en produisant des formules par calcul de plus faible précondition, nommées **obligations de preuve** (ou "condition de vérification"), qui sont ensuite injectées dans les prouveurs de théorèmes. Puis les prouveurs appliquent différentes techniques de résolution selon les théories sous-jacentes aux obligations de preuve.

Parmi les prouveurs on distingue les prouveurs interactifs, dont les plus cités sont COQ [COQ11, BC04, Ber08], ISABELLE [ISA11, NPW02], et HOL [HOL11, GM93] pour la logique d'ordre supérieur, et les prouveurs automatiques, comme Simplify [DNS05], ALT-ERGO [ERG11, CCKL07] et Z3 [dMB08].

Plusieurs travaux ont utilisé la puissance expressive de la logique pour établir la preuve de programmes [FM07, FM04, BCDL05]. Ils procèdent de la manière suivante :

1. On annote le code source du programme en y insérant des hypothèses h et des conclusions c de part et d'autre d'une action a pour former des formules de la logique de Hoare [Hoa69].
2. On calcule la plus faible précondition [Dij75] $wp(a, c)$ pour établir les obligations de preuve de la forme $h \Rightarrow wp(a, c)$ devant être vérifiées afin de prouver la correction du programme.
3. On appelle un prouveur de théorèmes pour prouver ces obligations de preuves.

.

La preuve interactive permet de prouver une large classe de propriétés sur les programmes mais requiert un utilisateur expert pour annoter les programmes et piloter la

preuve. Parmi les outils de vérification utilisant l'interaction avec l'utilisateur, on considère Boogie [BOO11, BCDL05] qui vérifie les programmes .NET annotés en Spec♯ et utilise le prouveur Z3 [dMB08], B4free qui prouve les modèles B [ALN+91] en utilisant un prouveur interactif appelé Balbulette [AC03] et ESC/Java [CK04, Kin11] qui vérifie les programmes Java annotés en JML et utilise le prouveur Simplify [DNS05].

ESC/Java2

ESC/Java2 (*Extended Static Checker*) [CK04, Kin11] est la suite des travaux de ESC/Java initialement menés par Compaq. Il s'agit d'une approche pragmatique et simple, qui vise à aider à la détection d'erreurs à l'exécution dans les programmes Java, tels que les déréférencements de pointeurs null, les divisions par zéro, ou le dépassement d'indices dans les tableaux. ESC/Java2, repris par KindSoftware [Kin11], intègre le support des annotations JML [LBR98, LBR99] et vérifie en plus la cohérence des annotations avec le programme en utilisant le prouveur de théorèmes Simplify [DNS05]. ESC/Java2 s'appuie sur des techniques de réfutation qui visent à trouver un contre-exemple.

À partir d'un programme annoté, ESC/Java2 génère des conditions de vérification qui représentent des conditions garantissant l'absence des erreurs précédemment citées. ESC/Java2 génère une condition de vérification par type d'erreur. Ces conditions sont ensuite déchargées dans le prouveur Simplify [DNS05]. Ce dernier décide alors si une condition de vérification est valide ou invalide et produit un contre-exemple dans le second cas. Ce dernier est ensuite analysé et rendu à l'utilisateur pour produire un avertissement relatif au type d'erreur considéré. Dans le cas contraire, c'est-à-dire si la condition de vérification est valide, le programme est déclaré exempt du type d'erreur considéré. ESC/Java2 n'est ni correct (*sound*) ni complet, mais il trouve beaucoup d'erreurs en pratique [CK04].

2.2.4 Frama-C

Frama-C est une plate-forme pour l'analyse statique des programmes C qui fournit un ensemble d'outils dont l'analyse de valeurs qui effectue une analyse statique des valeurs de variables d'un programme par interprétation abstraite et Jessie et WP qui implémentent des techniques de preuve. Comme Frama-C est utilisé pour les principales contributions de cette thèse, nous détaillons sa description pour le chapitre suivant, dans la section 3.4.

2.3 Analyse dynamique

La génération de tests est la technique la plus utilisée dans l'industrie pour la validation de programmes. Elle est définie dans [Ins90] de la manière suivante :

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

Le test consiste à stimuler un système en fournissant des entrées, en exécutant le programme sur ces entrées et en comparant le résultat obtenu au résultat attendu. Ce dernier est obtenu par un oracle qui permettra d'établir un verdict pour chaque test. L'oracle est par exemple fourni par un modèle du système à tester ou par le testeur lui-même.

Les tests peuvent s'appliquer durant la phase de développement du système. Ils sont généralement effectués au plus tôt dans le développement du logiciel de manière à minimiser l'impact d'éventuelles erreurs sur son coût de production. Ils perdurent tout au long de la vie d'un logiciel, même lorsque de nouvelles fonctionnalités sont ajoutées.

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

1. Tout d'abord on effectue des tests séparément sur les petites entités ou composants d'un système pour s'assurer que cette partie est correcte. Les tests de chaque entité sont indépendants les uns des autres. C'est l'étape de **test unitaire**.

2. Puis, les agrégats d'entités font l'objet de nouveaux tests qui visent à mettre en évidence l'existence d'éventuels défauts dans la communication des entités au sein des agrégats. C'est la phase de **test d'intégration** [Mye79, Bei90].

3. Lorsque toutes les entités du logiciel sont intégrées, on cherche à tester les fonctionnalités du logiciel complet pour s'assurer qu'elles correspondent bien au besoin de l'utilisateur final. C'est la phase de **test de validation**.

4. Enfin, si l'on ajoute de nouvelles fonctionnalités, il faudra s'assurer de leur bon fonctionnement mais également que cet ajout ne détériore pas les anciennes fonctionnalités. C'est la phase de **test de non régression**.

Ces grandes phases de test sont présentées dans de nombreux modèles représentant le cycle de vie de développement. Le plus connu de ces modèles est certainement le cycle de vie en "V" présenté dans la figure 2.1 où la branche descendante du cycle correspond aux phases de développement et la branche montante correspond aux tests effectués à chaque phase.

On distingue différentes techniques de génération de tests, présentant différents niveaux d'automatisation (manuelle, semi-automatique ou automatique), et différents niveaux de connaissance du programme testé (complète, partielle ou inconnue). Dans ce mémoire, nous nous intéressons uniquement aux techniques automatisées visant à dériver des tests

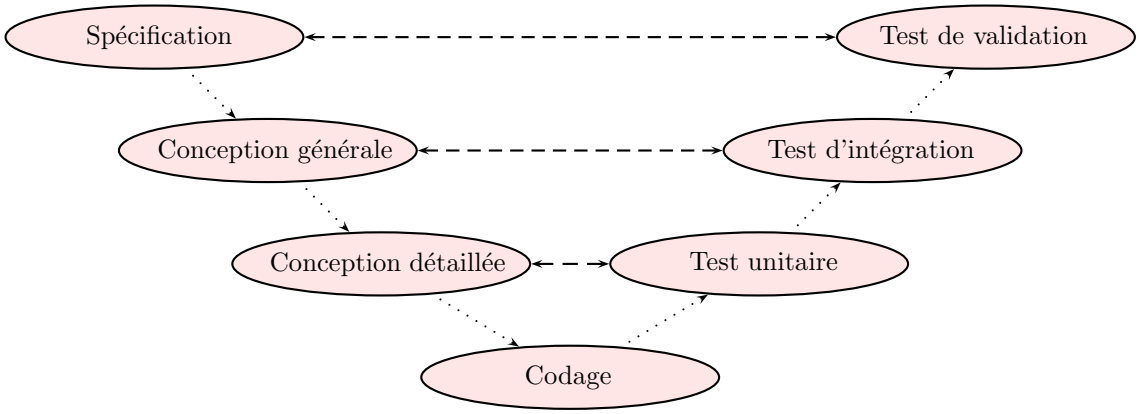


FIGURE 2.1 – Cycle de vie en V

à partir d'un programme.

En général, les techniques de test ne sont pas exhaustives et l'absence d'échecs lors du passage des tests n'est pas une garantie de bon fonctionnement du système. Néanmoins, selon les critères utilisés pour la génération des tests, et selon la couverture fournie par les tests, un système ainsi validé peut acquérir un certain niveau de confiance auprès de ses utilisateurs.

Les méthodes de test peuvent être classées en trois catégories : le test aléatoire, le test fonctionnel et le test structurel.

1. Le test aléatoire consiste à donner, au hasard, des valeurs aux entrées du programme. On parle de **jeu de test**. En exécutant de nombreux tirages, on peut ainsi atteindre une confiance relativement élevée dans la fonction. Cependant, si certaines zones sensibles ne sont atteintes qu'avec une probabilité faible par rapport aux valeurs d'entrée, ce genre de test aura du mal à les couvrir.

2. Le test fonctionnel génère les jeux de test en fonction des spécifications de la fonction testée. Des valeurs seront choisies pour chaque région du domaine d'entrée où le programme doit avoir un comportement particulier.

3. Le test structurel permet de déterminer des valeurs d'entrée par une analyse du code source de la fonction. On vise une couverture structurelle du graphe de flot de contrôle selon un critère de couverture choisi. Par exemple, on s'assure que toutes les branches possibles ont été couvertes (ou au moins un pourcentage élevé d'entre elles).

Cette partie détaille les techniques de génération de tests existantes qui sont en rapport avec les techniques citées et employées dans les travaux menés dans le cadre de cette thèse. Nous commençons par l'approche "structurelle". Pour celle-ci, nous présentons l'approche concolique. Si cette dernière technique est l'objet des travaux présentés

dans ce mémoire, les autres ne sont pas pour autant sans intérêt car elles pourraient également s'appliquer dans le cadre d'une combinaison analyse statique / analyse dynamique.

2.3.1 Test structurel

Le but d'un jeu de tests est de fournir des données d'entrée, pour une fonction testée, qui permettent de vérifier chaque comportement réel de cette fonction. Pour le test structurel, la notion de comportement est liée au code source de la fonction sous test. On distingue là encore deux types de test : le test orienté flot de contrôle et le test orienté flot de données. Le test orienté flot de données cherche à couvrir toutes les relations entre la définition d'une variable et son utilisation. Au contraire, le test orienté flot de contrôle s'intéresse plus à la structure du programme. On distingue trois critères de couverture.

1. Le critère "**toutes les instructions**" cherche à couvrir toutes les instructions du programme par au moins un cas de test.
2. Le critère "**toutes les branches**" cherche à couvrir les deux branches (vraie et fausse) de chaque instruction conditionnelle dans le programme.
3. Le critère "**tous les chemins**" cherche à exécuter tous les chemins du graphe de flot de contrôle du programme.

En pratique, le critère "tous les chemins" reste le meilleur critère pour trouver des bugs. Néanmoins le calcul des données nécessaires pour exécuter un chemin donné peut être coûteux. De plus le nombre des chemins croît exponentiellement avec le nombre d'instructions conditionnelles présentes dans le programme. Avec les boucles, ce nombre peut augmenter à l'infini, ce qui rend ce critère difficilement applicable aux programmes de grande taille. Le critère "**tous les k-chemins**" est une variante du critère "tous les chemins". Il limite la recherche de tous les chemins du graphe de flot de contrôle du programme aux chemins qui ont au plus k itérations consécutives des boucles.

L'exécution symbolique dynamique, aussi appelée exécution concolique, effectue simultanément une exécution concrète et une exécution symbolique dans un cadre de collaboration afin d'explorer les chemins du graphe de flot de contrôle. Dans ce cadre, l'execution symbolique sert à générer les données de tests et l'exécution concrète sert à guider l'exploration. L'exécution concrète permet aussi d'approximer la valeur d'expressions complexes lors de l'exécution symbolique telles que les expressions non linéaires par des valeurs concrètes obtenues lors de la dernière exécution concrète.

La méthode concolique et son implémentation dans l'outil PathCrawler seront détaillées dans la partie 3.3.4. Cette technique proposée initialement par [WMMR05] et [GKS05] est depuis exploitée par différentes méthodes de génération de données de test. Parmi les applications, on peut citer le test des programmes en C [BDHTH+09, GKS05, God07, SMA05], en bytecode Java [SA06, CG10], les programmes .Net [TdH08, TS06] et les binaires [GLM08, BH08]. Certaines méthodes sont spécialisées dans la recherche de bogues. Nous décrivons les méthodes suivantes : DART et son successeur SMART,

CUTE, JCUTE, PEX, EXE et son successeur KLEE et SAGE. Ces outils essaient de mettre en évidence des erreurs à l'exécution ou des failles de sécurité. Nous les présentons brièvement ci-dessous suivant l'étude de [Cha10].

DART / SMART

DART (Directed Automatic Random Testing) propose une méthode basée sur l'exécution concolique pour couvrir tous les chemins exécutables ou toutes les branches dans un programme C [GKS05]. Il parcourt le graphe de flot de contrôle en profondeur d'abord.

Selon l'état de l'art de [Cha10], DART fait des approximations sur les valeurs en utilisant des valeurs concrètes obtenue lors de la dernière exécution quand il ne peut pas résoudre symboliquement certains expressions. Notamment, dans le cas des expressions complexes portant sur les pointeurs qui ne sont pas supportées par son modèle mémoire.

[Cha10] note que le nombre de comportements du programme pris en considération par DART peut se trouver réduit par ces approximations. Par conséquent certains chemins exécutables peuvent ne pas être couverts par des données de test à cause de l'approximation.

SMART (Systematic Modular Automated Random Testing), le successeur de DART, étend la méthode DART avec un traitement spécifique des appels de fonction [God07] permettant d'optimiser l'analyse de ces fonctions appelées. A chaque fois qu'on analyse une fonction, on créée un résumé sous la forme d'un couple précondition-postcondition. Si la même fonction est appelée une deuxième fois dans le même contexte, Le résumé est utilisé directment et on n'a pas besoin d'analyser la fonction une deuxième fois. Ces résumés sont calculés à la demande, En d'autres termes, à chaque appel de fonction lors de l'exécution symbolique dynamique, on vérifie l'existence d'un résumé pour cette fonction avec le même contexte d'appel. Si oui, le résumé est utilisé directement, sinon, la fonction est analysée par exécution symbolique dynamique et on calcule et sauvegarde un nouveau résumé.

CUTE / JCUTE

La méthode CUTE [SMA05, SA06] propose de couvrir tous les chemins faisables d'une fonction C. JCUTE est l'implémentation de la méthode pour tester le bytecode Java. On peut limiter l'exploration en bornant la longueur maximale l des chemins. Les auteurs introduisent le terme **exécution concolique** pour désigner la coopération entre exécution concrète et exécution symbolique. Une première exécution concrète permet de construire un modèle symbolique de la fonction. Ensuite différentes exécutions symboliques de ce modèle permettent d'atteindre la couverture désirée.

CUTE utilise un solveur de contraintes pour les contraintes linéaires et explore en profondeur d'abord le graphe de flot de contrôle symbolique de la fonction sous test. Il fait des approximations sur les valeurs des variables de type pointeur, par des valeurs abstraites. Comme dans DART, Ces approximations peuvent mener, d'une part, à la génération de données de test visant à exécuter certains chemins non faisables, et, d'autre part, à la non exécution de certains chemins faisables.

PEX

PEX [TS06, TdH08] propose une méthode basée sur l'exécution symbolique dynamique pour couvrir toutes les branches. Il est développé par Microsoft et il est intégré à Visual Studio pour generer des tests pour les programmes C♯. Il utilise le solveur Z3 [dMB08] pour la résolution des contraintes.

Selon [Cha10], de multiples paramètres permettent de borner l'exploration tels que :

1. le nombre d'exécutions,
2. la longueur des chemins,
3. le nombre de branches pour un chemin,
4. le temps attribué à la génération,
5. le temps attribué au solveur pour un chemin,
6. le nombre de conditions portant sur les entrées pour un chemin,
7. la taille de la pile,
8. la taille de la mémoire.

Comme dans les méthodes utilisées dans DART, SMART, CUTE et JCUTE, PEX fait des approximations sur les valeurs des variables des expressions complexes, par des valeurs concrètes de la dernière exécution. Ces approximations peuvent mener à la non couverture de certaines branches. De plus si l'outil ne parvient pas à générer des entrées pour un préfixe exécutable, tous les chemins débutant par ce préfixe sont exclus de l'exploration.

EXE / KLEE

EXE [CGP+08] et son successeur KLEE [CDE08] sont des outils de déctection des erreurs à l'exécution pour les programmes C. Ils exploitent l'exécution symbolique dynamique pour couvrir tous les chemins dans un programme C. Pour chaque chemin sur lequel se trouve une instruction pouvant créer une erreur à l'exécution (overflow, division par zéro, etc), EXE et KLEE résolvent un système de contraintes pour créer une entrée

provoquant cette erreur. La couverture des chemins se fait selon une heuristique qui choisit en priorité des chemins couvrant des instructions non encore couvertes.

Selon [Cha10], l'approche suivie par les deux outils crée un processus pour chaque chemin partiel du programme, qu'il soit faisable ou non. En effet, quand une expression conditionnelle est symboliquement exécutée, un nouveau processus est créé (par duplication à l'aide de *fork*) afin qu'il y ait deux processus exécutants les deux branches de la conditionnelle. C'est à dire, le processus initial va cotinuer l'exécution dans une branche, et le nouveau processus créé va essayer de génére un cas de test qui suit le même prédicat (avant la conditionnelle) que le processus initial et qui va continuer dans la branche non couverte. La priorité donnée à chacun des processus détermine dans quel ordre les chemins sont parcourus.

EXE et KLEE utilisent le solveur STP [GD07] pour la résolution des contraintes. STP prend en entrées des formules sur la théorie des vecteurs de bits et des tableaux (Cette théorie permet de réprésenter la plupart des expressions des langages comme C / C + + / Java et Verilog). Il vérifie si la formule est satisfiable ou non. Si l'entrée est satisfaisable, STP génère également une affectation de variable satisfaisant la formule d'entrée. STP implante toutes les opérations arithmétiques sur les entiers et les booléens, y compris les opérations non linéaires.

La méthode ne traite les pointeurs que partiellement. Elle ne peut pas symboliquement traiter le double déréférencement (par exemple `**p`). En présence d'un double déréférencement `**p`, la méthode fait des approximations, elle approxime la première déréférence `*p` par une valeur concrète parmi ses valeurs possibles.

L'utilisation d'une duplication de processus est intéressante dans le cas où une parallélisation est possible (grille de calcul ou cloud). Contrairement aux autres méthodes utilisant l'exécution symbolique dynamique, cette approche ne cherche pas à détecter dès que possible les préfixes de chemin non exécutables. En effet, la résolution des contraintes n'est lancée que lorsqu'un chemin complet est parcouru. Cela peut ralentir considérablement le parcours des chemins en présence de chemins partageant le même préfixe de chemin non exécutable.

SAGE

SAGE [GLM08] (Scalable, Automated, Guided Execution) est un outil développé par Microsoft. Il propose aussi une méthode basée sur l'exécution symbolique dynamique pour couvrir tous les chemins exécutables d'un programme, mais dédiée à trouver rapidement des bogues dans des programmes de très grande taille.

SAGE génère des tests sur les binaires X86. Il utilise un critère d'exploration, dit générationnel, qui est adapté à l'exploration des programmes de grande taille ayant des chemins très profonds (des centaines de millions d'instructions). Il utilise une heuristique

pour maximiser la couverture de code le plus rapidement possible, afin de trouver des bogues plus vite.

SAGE a été appliqué à certaines applications Windows et il a détecté de nouveaux bogues. Parmi tous les outils présentés ici, SAGE est celui qui passe le mieux à l'échelle des applications de grande taille. Ce succès repose en grande partie sur son approche générationnelle mais aussi sur les ressources disponibles pour générer et exécuter les tests (des centaines de machines qui tournent pendant des mois).

2.3.2 Test fonctionnel

Dans cette approche du test, on étudie le comportement fonctionnel du programme. Le logiciel est vu comme une boîte noire. La seule information dont un testeur dispose est la manière dont cette boîte réagit aux stimuli externes. Le test fonctionnel est utilisé pour vérifier la conformité des réactions du logiciel avec les attentes des utilisateurs.

Il existe de nombreuses techniques qui se différencient essentiellement par la manière de choisir les données de test. On distingue :

1. Le **test de partition** : les valeurs d'entrées du logiciel sont partitionnées en classes d'équivalence, sur lequelles le logiciel doit avoir le même comportement (*domain splitting*). Une seule valeur aléatoire est choisie dans chaque classe de la partition.
2. Le **test aux limites** : on se base sur les bornes des domaines de définition des variables pour proposer des données de test. Il s'agit d'une méthode très efficace pour trouver les bogues.

GATeL

GATeL [MA00] est un outil de génération de tests fonctionnels développé au laboratoire de sûreté des logiciels au CEA LIST. Il génère automatiquement des séquences de tests à partir de spécifications décrites dans le langage LUSTRE [HCRP91]. Le langage LUSTRE est un langage déclaratif à flots de données synchrones muni d'opérateurs temporels permettant de se référer à des valeurs passées des données. GATel se base sur une représentation symbolique des états du système par un système de contraintes et construit les cas de test par chaînage arrière à partir de l'état cible. L'état cible est décrit à l'aide de contraintes qui peuvent contenir aussi bien des invariants que d'autres contraintes exprimées en LUSTRE. Ces dernières décrivent un objectif de test. Lors de la génération de test, GATeL recherche une séquence qui satisfasse à la fois l'invariant et l'objectif de test. Cet outil offre la possibilité de réaliser des partitions de domaines.

2.4 Simplification syntaxique

La transformation de programmes permet d'obtenir un nouveau code en opérant des modifications sur le code d'un programme existant. On s'intéresse particulièrement à la simplification du code.

La simplification effectue une coupe d'un programme afin d'en conserver les parties répondant à un critère préétabli. Il s'agit d'une modification purement structurelle. La technique la plus courante s'appelle le *slicing* [Wei81, Wei82].

Le *slicing* est une technique, originalement proposée par Mark Weiser [Wei81], qui permet de décomposer automatiquement un programme en analysant son flot de contrôle et son flot de données pour isoler des comportements satisfaisant un **critère de slicing**. Le résultat d'une telle décomposition, pour un programme et un critère donnés, est un sous-programme, nommé *slice*, qui possède le comportement original relativement aux éléments choisis par le critère de *slicing*.

Toutes les instructions et les branches du programme original qui n'ont pas d'intérêt par rapport au critère considéré n'apparaissent pas dans le sous-programme.

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

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

Cette méthode de *slicing* est intra-procédurale. Elle permet de créer la *slice* d'une entité indépendamment des autres entités appelées ou appelantes avec lesquelles elle pourrait être en interaction. La méthode proposée par Weiser pour étendre le *slicing* d'une fonction f à un *slicing* inter-procédural de f consiste à produire une *slice* comme l'union des *slices* intra-procédurales des fonctions appelantes et appelées par f. L'inconvénient de cette méthode inter-procédurale est qu'elle ne produit pas des *slices* précises. En effet, l'union des *slices* peine à prendre en considération le contexte d'appel des fonctions. [HRB90, SHR99] ont proposé une autre méthode de *slicing* inter-procédural. Cette méthode se base sur l'utilisation d'un graphe particulier : le *System Dependence Graph* [HRB90]. Ce graphe est une extension du *Program Dependence Graph* dont les travaux de Ottenstein et Ottenstein [OO84] ont montré l'adéquation avec la technique de *slicing*.

Le *System Dependence Graph* permet de représenter des programmes dans lesquels on trouve des appels de fonctions tandis que le *Program Dependence Graph* ne permet qu'une représentation des programmes constitués d'une seule fonction. Le *System Dependence Graph* est constitué d'un *Program Dependence Graph* pour la fonction principale

du programme et de *Procedure Dependence Graphs* pour toutes les autres fonctions du programme. Le lien entre ces différents graphes est réalisé grâce à des arcs spécifiques. On distingue deux types d'arcs :

1. Les arcs qui représentent une dépendance directe entre le lieu de l'appel et la fonction appelée.
2. Les arcs qui représentent une dépendance transitive due aux appels de fonctions.

De plus de nouveaux nœuds sont introduits avant et après chaque graphe de dépendance afin de représenter :

1. L'entrée dans la fonction appelée.
2. Le passage de paramètres. Un nœud temporaire est utilisé pour chaque paramètre de façon à mettre en relation les paramètres effectifs de l'appel et les paramètres formels de la procédure appelée.
3. La même mise en relation est effectuée pour le renvoi des résultats à la fonction appelante lorsque c'est nécessaire.

Ces nœuds sont reliés entre eux par des arcs spécifiques.

L'outil de *slicing* fourni par Frama-C se base sur ces travaux.

2.5 Combinaisons de méthodes dans différents outils

Récemment, plusieurs méthodes et outils proposent des combinaisons d'analyses statiques et dynamiques pour la vérification des programmes. [Ern03] présente diverses manières de combiner les deux techniques pour les mettre en synergie. Nous présentons maintenant différents outils combinant les deux types de méthodes.

2.5.1 SLAM

SLAM [SLA11, BR02] (*Software (Specifications), Languages, Analysis, and Model checking*) est un outil développé par Microsoft, initialement conçu pour la validation des propriétés des drivers Windows XP. Il fut par la suite utilisé pour la vérification des propriétés de sûreté de programmes C. La vérification se base sur l'abstraction par prédicats [GS97] : afin de montrer qu'une propriété (de sûreté) est satisfaite par un système, il suffit de montrer qu'elle est satisfaite par une abstraction du système. À partir d'un ensemble de prédicats et invariants du système fournis par l'utilisateur, SLAM construit une abstraction. Ensuite, si l'outil ne peut pas établir que la propriété est satisfaite, en exploitant les contre-exemples générés par l'étape de model-checking, l'abstraction est raffinée. Ainsi l'analyse se poursuit par une boucle répétant les opérations :

1. abstraire,

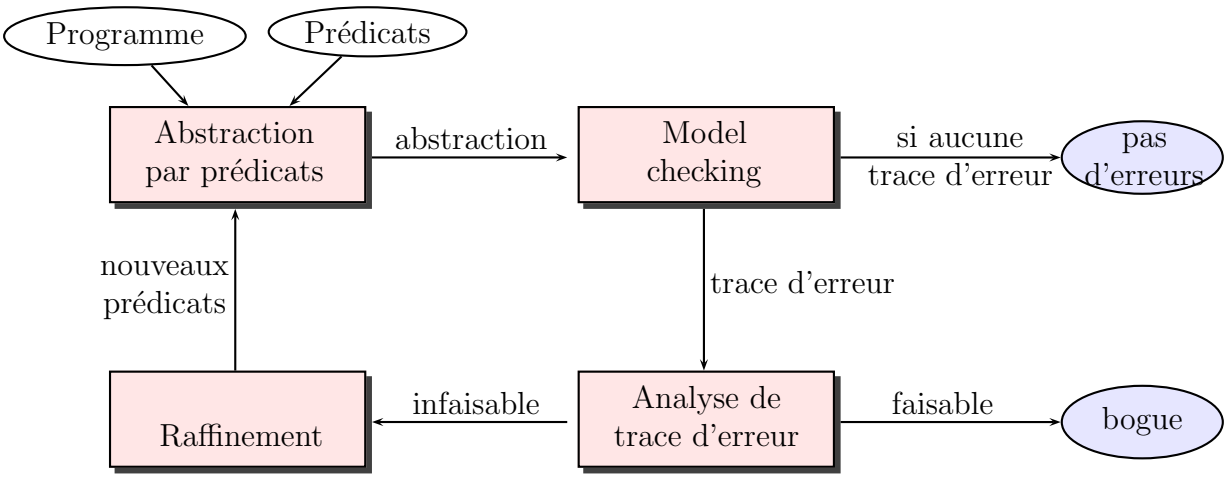


FIGURE 2.2 – Raffinement d'abstraction guidé par les contre-exemples

2. vérifier,
3. raffiner,

jusqu'à conclure sur la propriété. Le processus peut ne pas terminer.

2.5.2 Raffinement d'abstraction guidé par les contre-exemples

Le raffinement d'abstraction guidé par les contre-exemples [CGJ+00] (*Counter-Example Guided Abstraction Refinement* ou CEGAR) effectue un raffinement de l'abstraction à partir de prédicats. Il combine abstraction, model-checking et preuve. Comme nous l'avons déjà mentionné, lorsque l'abstraction ne contient pas d'erreurs de sûreté, le programme ne contient pas d'erreurs non plus. Cependant, quand il y a une trace d'erreur dans l'abstraction, il n'est pas sûr que ce soit une véritable erreur étant donné la sur-approximation calculée par l'abstraction à partir de prédicats. Pour éliminer les fausses alarmes, il faut raffiner l'abstraction.

CEGAR présente une technique pour le raffinement systématique de l'abstraction. Étant donné un ensemble initial de prédicats, une abstraction du programme est construite en utilisant des techniques de preuves selon les méthodes précédemment présentées (section 2.2.2). Ensuite, l'abstraction est raffinée itérativement en se basant sur les traces d'erreur abstraites. Le processus est illustré par la figure 2.2. Dans chaque itération :

1. Une abstraction plus fine est créée.
2. Ensuite, les techniques de model-checking sont utilisées pour calculer tous les chemins abstraits d'exécution. Si l'abstraction ne contient pas de chemins abstraits d'erreur, alors le programme ne contient pas d'erreurs. Dans le cas contraire, le vérificateur de modèles fournit une trace abstraite d'erreur. Ce chemin correspond soit à un vrai bogue, soit à une fausse alarme.

3. On analyse la trace pour savoir si elle est faisable dans le programme initial. Si la trace est faisable, alors elle représente une trace d'erreur réelle dans le programme initial. On signale un bogue.
4. Si la trace n'est pas faisable, on raffine l'abstraction. L'étape de raffinement ajoute de nouveaux prédicats pour éliminer cette trace infaisable de l'abstraction raffinée. Les nouveaux prédicats ajoutés sont déduits à partir d'une analyse de l'infaisabilité de la trace d'erreur.
5. On répète les actions de 1 à 5.

L'algorithme peut ne pas terminer.

Plusieurs outils de vérification des programmes C se basent sur ces méthodes, parmi lesquels BLAST [BLA11, BHJM05, BHJM07], YOGI [GHK+06, NRTT09] et MAGIC [CCG03].

2.5.3 BLAST

Dans BLAST [BLA11] (*Berkeley Lazy Abstraction Software Verification Tool*), la vérification est effectuée par des abstractions et des raffinements guidés par des contre-exemples. BLAST est utilisé pour la vérification de propriétés comportementales de programmes C. À chaque étape de la boucle, il réutilise le résultat de l'étape précédente de la boucle. Le raffinement de l'abstraction n'est pas effectué sur l'ensemble de l'espace d'état, mais seulement sur la partie où c'est nécessaire. Ainsi cette méthode de raffinement local est moins coûteuse car elle limite le nombre d'états.

Dans [BHJM07], les auteurs utilisent l'outil CCured [NCH+05] pour générer des assertions avant chaque accès par des pointeurs ou dans des tableaux pour s'assurer que la case mémoire accédée est valide. Ces assertions sont ensuite analysées par BLAST avec comme objectif de détecter une erreur violant ces assertions ou de prouver qu'aucune exécution ne permet d'atteindre un état d'erreur.

Ces travaux sont très proches des nôtres. Dans la combinaison CCured-Blast, BLAST doit vérifier tous les accès mémoires tandis que SANTE analyse uniquement les alarmes signalées par l'analyse de valeurs. Contrairement à SANTE, BLAST n'effectue pas de simplifications syntaxiques par le *slicing*.

2.5.4 YOGI

Comme BLAST, YOGI [GHK+06, NRTT09] implémente l'algorithme DASH [BNR+10], initialement appelé Synergy [GHK+06], combinant le test et la vérification dans le but de déterminer avec certitude si un programme satisfait une propriété ou non. La propriété à vérifier est de type sûreté/atteignabilité, et correspond à un ensemble d'états d'erreur que le programme ne devrait pas atteindre.

L'objectif est soit d'obtenir un contre-exemple atteignant un état d'erreur, soit de prouver qu'aucune exécution du programme ne permet d'atteindre un tel état. L'algorithme peut ne pas se terminer, mais la terminaison garantit un résultat sûr.

L'éventuelle preuve de sûreté du programme est obtenue sous la forme d'une abstraction finie du programme ne contenant aucun chemin abstrait permettant d'aboutir à un état d'erreur.

Ces travaux sont très proches des nôtres. YOGI traque les états d'erreur comme SANTE. Dans SANTE, nous utilisons l'analyse de valeur alors que YOGI utilise l'abstraction à partir de prédicats. Dans YOGI, les états d'erreurs sont spécifiés comme une propriété d'entrée alors que dans SANTE ils sont automatiquement calculés par l'analyse de valeurs. YOGI n'effectue pas de simplifications par *slicing*. Leur approche est plus adaptée pour vérifier une propriété à la fois, tandis que SANTE peut être utilisé sur plusieurs alarmes simultanément.

2.5.5 Daikon

La génération de formules valides à des emplacements spécifiques du programme (invariants) peut être utilisée pour améliorer les résultats des techniques de vérification. Par conséquent, les techniques qui génèrent ces invariants peuvent être utilisées comme une première étape, en combinaison avec d'autres techniques qui utilisent ces invariants. Par exemple, les invariants peuvent être utilisés comme précondition, postcondition ou invariant de boucle pour la vérification déductive. Daikon [EPG+07] est l'un des outils les plus connus, utilisant la génération des invariants.

Daikon [EPG+07] est un système de détection des candidats d'invariants dans un programme à partir de la génération de tests. Un invariant est une propriété qui est toujours vraie à un point de programme comme par exemple : $x > 0$, $y = 10$, et $p = null$. Daikon exécute le programme pour recueillir des informations pertinentes sur le programme.

Daikon est composé de deux parties. La première, le *frontend*, est spécifique au langage utilisé. Elle extrait les informations d'état du programme en cours. L'état se compose des variables et de leurs valeurs. Pour extraire ces informations, Daikon ajoute des instructions de collecte d'informations dans le programme pour observer les informations pertinentes lors de l'exécution dans un fichier de trace. Daikon fournit des interfaces pour les langages C, Java et Perl. Cette partie s'appelle Kvasir [Guo06] dans le cas de C et C++. La deuxième partie correspond au moteur d'inférence indépendant du langage.

Le moteur d'inférence d'invariant utilise des techniques d'apprentissage [ECGN01] pour détecter les invariants. Les formules détectées sont des candidats d'invariants, parce qu'elles sont déduites de l'ensemble des états observés, qui est un sous-ensemble de l'ensemble de tous les états possibles. Par conséquent, ce qui a été observé comme invariant peut être une variante dans certaines exécutions. En d'autres termes, l'invariant n'est vrai

que pour les exécutions observées et il ne l'est peut-être pas pour celles qui ne sont pas observées. Ainsi, la qualité des résultats dépend des tests générés. La sortie du moteur d'inférence est l'ensemble des invariants détectés.

2.5.6 Check 'n' crash

L'outil Check 'n' Crash [CS05] utilise la génération de tests pour filtrer les fausses alarmes parmi les alarmes signalées par l'analyse statique. Check 'n' crash utilise ESC/-Java pour analyser statiquement le programme et détecter les bogues potentiels. Ensuite le générateur de test JCrasher [CS04] est utilisé pour confirmer ces alarmes.

L'outil n'a pas atteint le niveau désiré de rentabilité. En particulier, les utilisateurs se plaignent du travail d'annotation qui est considéré comme une lourde tâche, et d'un taux élevé de détection de fausses alarmes, surtout sur les programmes non annotés ou partiellement annotés. SANTE utilise une analyse de valeurs interprocédurale qui nécessite seulement une précondition, et la génération de tests de tous les chemins peut directement révéler que certaines alarmes sont inatteignables.

2.5.7 DSD Crasher

DSD Crasher [CS06] (*Dynamic inference, Static analysis, and Dynamic verification*) introduit une nouvelle phase d'analyse dynamique avant la combinaison (analyse statique puis analyse dynamique) utilisée dans Check 'n' Crash [CS05]. Il utilise Daikon pour fournir des hypothèses supplémentaires pour Check 'n' Crash. Ces hypothèses aident l'outil pour générer des tests et pour vérifier les alarmes signalées par ESC/Java.

DSD-Crasher exécute les actions suivantes :

1. Il exécute d'abord une suite de tests de non régression sur une application et génère ensuite des invariants en utilisant une version modifiée de Daikon. Les résultats de Daikon sont des annotations JML. Ces invariants sont alors utilisés pour guider le processus de raisonnement de Check 'n' Crash.

2. Check 'n' Crash utilise l'outil ESC/Java pour analyser statiquement le programme avec les invariants trouvés dans l'étape précédente. ESC/Java signale des erreurs potentielles.

3. Check 'n' Crash utilise le générateur de test JCrasher pour produire et exécuter des cas de tests.

Cette méthode génère des invariants incorrects qui peuvent conduire à classer sûres (*safe*) quelques erreurs réelles. Les auteurs disent : "*We are willing to trade some potential bugs for a lower false positive rate*". En comparaison, SANTE est correct et ne classe jamais

une erreur réelle (*bug*) comme *safe*.

2.6 Synthèse

Nous avons présenté dans ce chapitre diverses techniques de vérification d'analyse statique et d'analyse dynamique pour la validation de programmes. Nous avons également introduit les techniques de transformation de programmes, en particulier le *slicing*. Puis nous avons présenté un ensemble d'outils combinant ces techniques.

L'originalité de nos travaux présentés dans ce mémoire découlent d'une part de la combinaison de l'analyse statique et de l'analyse dynamique et d'autre part de l'ajout d'une étape de *slicing* déclinée de différentes manières. Cette étape permet de repousser les limites liées au manque de ressources des approches présentées. De plus, les erreurs sont signalées avec des informations plus précises et sont illustrées sur des programmes plus simples, ce qui n'est pas le cas avec les approches présentées. À notre connaissance, la combinaison de l'analyse de valeurs, du *slicing* et de la génération de tests pour la validation de programmes C n'a pas été préalablement étudiées.

Toutes ces étapes sont définies dans les chapitres relatifs aux contributions, en partie II. Mais avant d'y arriver, nous présentons dans le chapitre suivant les contextes initiaux de cette thèse, à savoir les deux outils PathCrawler et Frama-C ainsi que les services fournis par chaque outil. Nous nous sommes servis de ces outils pour mettre en œuvre notre approche.

Chapitre 3

Frama-C et PathCrawler

Sommaire

3.1 Introduction

Ce chapitre introduit et détaille le fonctionnement de l'outil PathCrawler de génération de tests structurels, et la plate-forme Frama-C pour l'analyse statique des programmes C. Par la suite, nous allons nous servir de ces deux outils pour implémenter notre méthode combinant l'analyse statique, le *slicing* et l'analyse dynamique.

Après la présentation de la méthode sante dans le chapitre 4, nous montrons dans le chapitre 5 l'adaptation et l'intégration de ces deux outils pour l'implémentation de la méthode.

Ce chapitre est organisé comme suit. Dans la partie 3.2, nous commençons par la présentation de la bibliothèque CIL (*C Intermediate Language*) [CIL06] sur laquelle se basent

```
1 int cond(int x, int y) {
2   if(x>0 && x<10){
3     y++;
4   }else{
5     y--;
6   }
7   return (y);
8 }
```

a) Programme initial

```
1 int cond(int x, int y) {
2   if (x > 0) {
3     if (x < 10) {
4       y++;
5     }
6     else {
7       goto _LAND;
8     }
9   }
10  else {
11    _LAND:
12    y--;
13  }
14  return (y);
15 }
```

b) Programme normalisé

FIGURE 3.1 – Normalisation du code avec CIL : décomposition des conditions multiples.

les deux outils. Dans la partie 3.3, nous présentons l'outil PATHCRAWLER. La partie 3.4 est dévolue à la présentation de la plate-forme FRAMA-C.

3.2 CIL

CIL est un analyseur syntaxique de code C. Il fournit un arbre de syntaxe abstraite (*Abstract Syntax Tree* ou AST) du programme analysé. C'est aussi une bibliothèque de fonctions qui permettent de manipuler cet arbre. Il fournit également des visiteurs qui permettent de consulter ou modifier les nœuds de cet arbre syntaxique. L'objectif principal de CIL est de faciliter l'analyse et la transformation des programmes C.

3.2.1 Normalisation de code par CIL

CIL effectue des transformations basiques sur le code. Ces transformations correspondent à une normalisation du langage C. Le programme résultant est un programme équivalent au programme initial mais normalisé. Les principales transformations effectuées par CIL sont :

1. la décomposition des conditions multiples,
2. le déplacement de toutes les déclarations des variables locales vers le début de fonction,
3. la mise sous une forme canonique des structures itératives,

```
1 int conflitVar(int b) {
2   int a;
3   int i;
4   int res;
5   a = 1;
6   if(b < 3){
7     float a = 0;
8     a = 2 * i;
9   }
10  res = 2 * a;
11  return (res);
12 }
```

a) Programme initial

```
1 int conflitVar(int b) {
2   int a;
3   int i;
4   int res;
5   float a_0;
6   a = 1;
7   if(b < 3){
8     a_0 = (float)0;
9     a_0 = (float)(2 * i);
10  }
11  res = 2 * a;
12  return (res);
13 }
```

b) Programme normalisé

FIGURE 3.2 – Normalisation du code avec CIL : renommage des variables.

4. l'isolation des instructions à effets de bord.

CIL exécute aussi d'autres transformations telles que la transformation des `switch` en des instructions conditionnelles imbriquées ou l'ajout des conversions (*cast*) explicites dans les endroits où le compilateur doit les ajouter.

Décomposition des conditions multiples

Les conditions multiples sont les conditions composées de conjonction et / ou disjonction de conditions élémentaires. La transformation, présentée dans la figure 3.1, est la décomposition des conditions multiples en une succession de conditions élémentaires. La figure 3.1a contient le code d'une fonction C contenant des instructions conditionnelles à conditions multiples. L'équivalent obtenu après normalisation est présenté dans la sous-figure 3.1b. La condition multiple conjonctive de la ligne 2 du programme initial est décomposée en deux conditions imbriquées (lignes 2 et 3 du programme normalisé).

CIL peut ajouter des étiquettes comme le symbole `_LAND` dans l'exemple 3.1b qui marque la branche "faux" de la condition initiale. En fait, pour une condition multiple conjonctive, il suffit qu'un unique membre ne soit pas vérifié pour que la condition ne le soit pas non plus. Nous nous apercevons que la transformation de cette condition multiple correspond à plusieurs conditions imbriquées. Dans la branche "faux" de chaque condition, il y a un `goto _LAND;`.

```
1 void boucles(int a, int b) {
2   int i;
3   for(i = 0; i < 10; i++) {
4     a = a + b;
5     b = a * b;
6   }
7 }
```

a) Programme initial

```
1 void boucles(int a, int b) {
2   int i;
3   i = 0;
4   while(1) {
5     while_0_continue: ;
6     if(i >= 10) {
7       goto while_0_break;
8     }
9     a = a + b;
10    b = a * b;
11    i++;
12  }
13  while_0_break: ;
14  return ;
15 }
```

b) Programme normalisé

FIGURE 3.3 – Normalisation du code avec CIL : forme canonique des structures itératives.

Remonter toutes les déclarations

Une autre transformation consiste à remonter toutes les déclarations de variables en début de fonction afin d'éliminer les conflits entre les variables locales d'une fonction d'une part et les variables internes à un bloc de la fonction ou les variables globales du programme d'autre part. CIL renomme certaines variables locales qui sont concernées par des conflits. Dans l'exemple 3.2a, les deux variables de même nom `a` sont déclarées à deux endroits différents de la fonction (lignes 2 et 7). Dans la version normalisée (exemple 3.2b), la deuxième déclaration est renommée en `a_0` (ligne 5). De plus, toutes les déclarations de variables locales sont remontées en début de fonction alors que, dans la fonction initiale, les déclarations locales étaient dispersées dans le corps de la fonction.

Mise sous une forme canonique des structures itératives

Une autre transformation est la mise sous une forme canonique des structures itératives. Cette transformation est illustrée par la figure 3.3. Le programme initial présenté dans la figure 3.3a de la ligne 3 à la ligne 6 contient une structure itérative. La figure 3.3b présente le programme normalisé. Toutes les structures itératives (`while`, `for`, `do .. while`), sont transformées en une structure itérative infinie `while(1)` (ligne 4) contenant une instruction de sortie de boucle (ligne 6) dont la condition est la négation de celle de la structure itérative initiale. La satisfaction de cette condition entraîne un saut juste après la fin de la structure itérative via l'utilisation de l'instruction `goto` (ligne 7). La non satisfaction entraîne l'exécution du corps de la structure itérative. Automatiquement deux étiquettes sont ajoutées pour chaque structure itérative (`while_0_continue`

```
1 int effetsDeBord(int a) {
2   return (a++ + f(a));
3 }
```

a) Programme initial

```
1 int effetsDeBord(int a) {
2   int tmp;
3   int tmp_0;
4   int tmp_1;
5   tmp = a;
6   a++;
7   tmp_0 = f(a);
8   tmp_1 = tmp + tmp_0;
9   return (tmp_1);
10 }
```

b) Programme normalisé

FIGURE 3.4 – Normalisation du code avec CIL : isolement des expressions à effets de bord.

et `while_0_break`). L'étiquette `while_0_continue` marque le début de la structure. Une instruction `continue;` aurait été remplacée par `goto while_0_continue`. L'étiquette `while_0_break` marque la sortie de la structure.

Isolation des instructions à effets de bord

Une autre transformation isole les instructions à effets de bord en ajoutant des nouvelles variables. Dans le code normalisé, les effets de bord ne sont pas autorisés à l'intérieur des expressions. La ligne 2 de la figure 3.4a contient une instruction qui fait la somme de deux expressions à effets de bord : `a++` et `f(a)`. Dans la sous-figure 3.4b, cette instruction est décomposée en deux instructions contenant chacune une expression à effet de bord (lignes 6 et 7). Cette transformation rend les effets de bord identifiables et permet de connaître la valeur d'une variable à chaque instant.

3.3 PathCrawler

Cette partie est dédiée à PATHCRAWLER. Avant la description générale en section 3.3.2, nous commençons par des préliminaires en section 3.3.1. Ensuite nous présentons les deux étapes de fonctionnement de PATHCRAWLER avant modification par nos travaux présentés dans ce manuscrit. L'étape d'analyse et instrumentation est détaillée en section 3.3.3. L'étape de génération est présentée en section 3.3.4.

3.3.1 Préliminaires

Dans cette section, nous introduisons le vocabulaire de la génération de tests et nous l'illustrons sur un exemple.

Le **test structurel** est une technique de test qui fonde la détermination des différents cas de test sur une étude de la structure interne de la fonction, c'est-à-dire la structure de son code.

Le **flot de contrôle** est l'ordre dans lequel les instructions d'un programme impératif sont exécutées. Il est défini par les différentes structures de composition d'instructions notamment conditionnelles ou itératives rencontrées au cours d'une exécution donnée.

Le **graphe de flot de contrôle** ou **CFG** (*Control Flow Graph*) d'un programme est un graphe connexe orienté avec un unique nœud d'entrée et un unique nœud de sortie, dont les nœuds sont les blocs de base du programme et les arcs représentent l'ensemble des branchements introduisant un nouveau flot de contrôle dans le programme. On entend ici par branchement, soit une structure conditionnelle soit une structure itérative.

Un **chemin d'exécution** est un chemin du CFG défini par une séquence de nœuds et d'arcs rencontrés lors d'un parcours du graphe de flot de contrôle de son entrée à sa sortie.

Un **chemin d'exécution partiel** est une séquence de nœuds et d'arcs rencontrés lors d'un parcours du graphe de flot de contrôle en partant de l'entrée sans atteindre forcément la sortie.

Un **prédicat d'un chemin** (ou condition de chemin) est une formule logique ψ sur les entrées du programme telle que l'exécution du programme sur une entrée V suit le chemin si et seulement si V vérifie ψ.

Un chemin d'exécution est dit **faisable** si son prédicat de chemin est satisfaisable, c'est-à-dire possède une solution. Dans le cas contraire, on dit que le chemin est **infaisable**.

Un **cas de test** d'une fonction est le couple des vecteurs de valeurs d'entrée injectées à la fonction et de sorties obtenues après exécution du programme sous test (si celui-ci termine).

Un **lanceur** est un composant logiciel qui appelle une entité du système sous test.

La **précondition** d'une fonction est un ensemble de propriétés portant sur les variables d'entrée (domaine de définition de chaque variable et relations entre ces variables).

Un **oracle** est une fonction qui évalue le résultat d'un cas de test.

Un **bogue** (*bug*) est la cause d'une erreur lors de l'exécution d'un programme.

3.3.2 Description générale

Développé au laboratoire de sûreté des logiciels (LSL) du CEA LIST, PathCrawler [WMM03, WMMR05, BDHTH+09] est un outil de génération automatique de tests pour les programmes C transformés au format CIL. Il prend en entrée le programme à tester avec sa précondition et fournit en sortie, un ensemble de tests couvrant l'ensemble des chemins d'exécution possibles avec le chemin parcouru par chaque cas de test. La méthode PathCrawler est souvent appelée **concolique** (concrète et symbolique) parce qu'elle effectue simultanément une exécution concrète et une exécution symbolique. Elle procède ainsi :

1. Les données d'un premier cas de test sont choisies aléatoirement,
2. le programme sous test est exécuté concrètement à partir des données de test sélectionnées et le chemin exécuté par ce cas de test est mémorisé sous la forme de son prédicat de chemin,
3. un nouveau chemin partiel à couvrir et son prédicat de chemin sont calculés suivant la stratégie de parcours en profondeur d'abord, en niant la dernière condition,
4. la programmation en logique par contraintes est utilisée pour évaluer la satisfaisabilité du nouveau prédicat de chemin et trouver un nouveau cas de test,
5. ensuite on répète les étapes 2, 3, 4 jusqu'à atteindre la couverture souhaitée.

Dans PathCrawler, on distingue deux critères de couverture :

1. Le critère **tous les chemins** cherche à exécuter tous les chemins du graphe de flot de contrôle du programme.
2. Le critère k**-chemins** cherche à exécuter tous les chemins du graphe de flot de contrôle du programme avec au plus k itérations des boucles.

Le fonctionnement de PathCrawler se découpe en deux grandes étapes (voir figure 3.5) qui correspondent essentiellement aux deux principaux modules du logiciel PathCrawler : d'une part, l'analyse et l'instrumentation du code et, d'autre part, la génération des cas de test. Dans la première étape, PathCrawler analyse le programme et la précondition et génère des fichiers nécessaires pour l'étape 2. Dans la deuxième étape, PathCrawler utilise les fichiers intermédiaires (issus de la première étape) pour générer les cas de tests. Les sections 3.3.3 et 3.3.4 décrivent ces deux étapes.

3.3.3 Étape 1 : Analyse et instrumentation du programme sous test

L'objet de cette étape est de préparer l'exécution symbolique et l'exécution concrète tracée du programme sous test lors de l'étape 2. PathCrawler analyse syntaxiquement le programme afin de le transformer et de générer des fichiers intermédiaires. Cette étape se décompose elle-même en quatre phases :

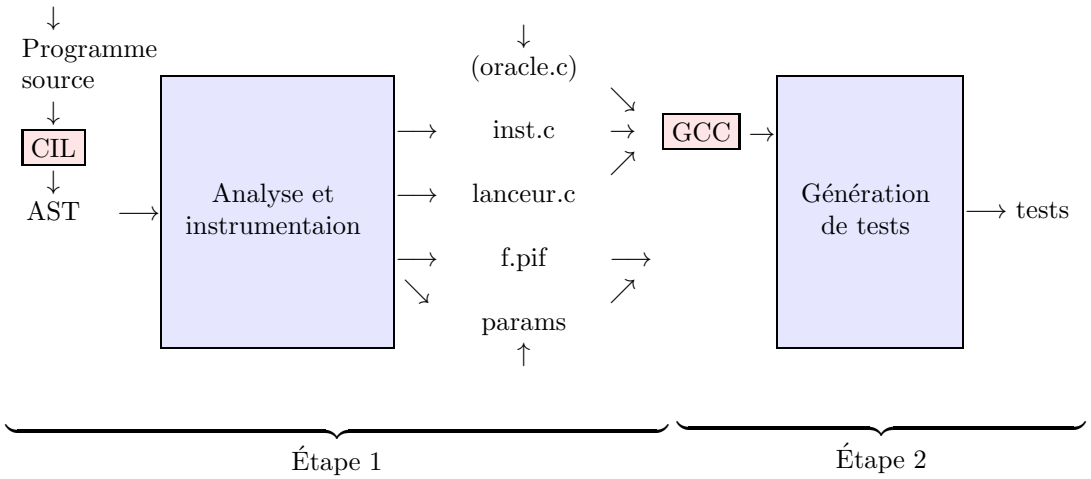


FIGURE 3.5 – Architecture de PATHCRAWLER avant modification par nos travaux présentés dans ce manuscrit

1. Analyse syntaxique du programme et construction de l'AST. Cette phase sert aussi à normaliser le code,
2. Préparation de l'exécution symbolique : traduction du programme en format intermédiaire de PATHCRAWLER (PIF),
3. Préparation de l'exécution concrète tracée : instrumentation du programme en injectant des instructions de traçage dans le programme normalisé,
4. Préparation du lanceur ou harnais de test : le lanceur exécute les cas de test sur le programme instrumenté.

Dans ce qui suit, nous détaillons chaque phase.

Analyse syntaxique du programme et construction de l'AST

Cette phase utilise l'analyseur syntaxique CIL [CIL06] pour parser et normaliser le code. CIL transforme le programme en un programme équivalent normalisé et fournit un arbre de syntaxe abstraite (AST) du programme normalisé.

Préparation de l'exécution symbolique

Lors de cette phase, on génère les fichiers intermédiaires à partir de l'AST normalisé fourni par CIL. Cette étape génère des clauses *Prolog* qui correspondent aux constructions de CIL, qui elles correspondent au programme C. La génération des fichiers intermédiaires est programmée en Ocaml en utilisant les visiteurs fournis par CIL.

Pour simplifier la présentation, nous avons représenté dans la figure 3.5 par un seul fichier `f.pif` l'ensemble des données qui contiennent une traduction du programme en

```
1 block(+139,cond(sup,     'x', 0),  pos, (2,1,'f.c') ).
2 block(-139,cond(infegal, 'x', 0),  neg, (2,1,'f.c') ).
3 block(+140,cond(inf,     'x', 10), pos, (2,2,'f.c') ).
4 block(-140,cond(supegal, 'x', 10), neg, (2,2,'f.c') ).
5 block(141,[affect('y', +('y', 1), (3,'f.c') )]).
6 block(144,[affect('y', -('y', 1), (5,'f.c') )]).
7 block(145,[return(y, (7,'f.c') )]).
```

FIGURE 3.6 – Format PIF généré pour le programme de la figure 3.1a

format intermédiaire (PIF). L'objectif de ce format intermédiaire est de représenter le programme dans une forme adaptée à la génération de tests par une méthode basée sur la programmation en logique par contraintes.

Chaque bloc élémentaire de condition ou d'affectation est identifié par un entier unique, appelé "**identifiant de bloc**". Dans la ligne 5 de la figure 3.6, l'entier `141` identifie le bloc constitué de l'affectation `y++`. Ces blocs sont réécrits sous forme de termes *Prolog* qui associent les identifiants aux blocs élémentaires afin de modéliser le chemin suivi dans un système de contraintes logiques du premier ordre. Le terme *Prolog* correspondant au bloc `141` est celui de la ligne 5 figure 3.6.

Les termes *Prolog* générés sont de la forme :

- *block(id, [termes])* pour les blocs élémentaires d'affectations, où *id* est l'identifiant du bloc et *termes* est l'ensemble des termes *Prolog* équivalent aux affectations du bloc. Un terme peut aussi être un appel de fonction ou un *return*.

- *block(id, cond(clause), pos/neg, (ligne, numero, fichier))* pour les branches des conditions avec :
  - *id* : identifiant du bloc de la branche, composé de l'identifiant du bloc de condition avec un "`+`" pour la branche "Vrai" ou un "`-`" pour la branche "Faux",
  - *clause* : condition encodée en *Prolog*,
  - *pos* si l'identifiant correspond à la branche "Vrai", et *neg* sinon,
  - *ligne* : numéro de la ligne dans le programme initial (avant la normalisation),
  - *numéro* : numéro de cette condition parmi les conditions de la ligne *ligne* utile dans le cas de conditions multiples,
  - *fichier* : nom du fichier source.

La figure 3.6 présente une version simplifiée du fichier PIF principal généré pour le programme de la figure 3.1. Les lignes 1 et 2 présentent les deux branches de la première sous-condition (`x > 0`) de la condition multiple à la ligne 2 du programme initial. La deuxième sous-condition de la même condition est présentée par les lignes 3 et 4. La ligne 5 représente le bloc qui contient l'affectation à la ligne 3 du programme initial (`y++`). La ligne 7 représente le retour de la fonction (`y`).

```
1 input_vals('cond', Inputs):-
2   input_val(x, int([0..100]), Inputs),
3   input_val(y, int([50..150]), Inputs),
4   true.
```

FIGURE 3.7 – Exemple de précondition (fichier `params`)

La précondition par défaut

Lors de cette étape, on génère également le fichier de la précondition par défaut. Il est représenté dans la figure 3.5 sous le nom `params`. Le fichier `params` est un fichier PIF qui décrit les domaines des entrées de la fonction sous test et les éventuelles relations entre ces entrées. Il peut être fourni par l'utilisateur. S'il n'est pas fourni par l'utilisateur, une précondition par défaut est générée et peut être ensuite ajustée par l'utilisateur.

Un exemple de précondition pour l'exemple de la figure 3.1 est donné dans la figure 3.7. Cet exemple définit les domaines des entrées de la fonction `cond`. La ligne 2 précise que `x` peut prendre des valeurs entières entre 0 et 100.

PIF contient d'autres commandes qui permettent de spécifier le domaine pour la taille d'un tableau ou pour le nombre d'éléments dans un bloc mémoire référencé par un pointeur ainsi que le domaine des éléments. On peut également définir des relations entre les entrées, par exemple qu'un tableau est trié ou qu'une variable `x` est supérieure à une autre variable `y`, etc.

Préparation de l'exécution concrète tracée

Cette étape, aussi appelée instrumentation, nous permet d'ajouter les instructions de trace nécessaires pour la modélisation du chemin suivi par un système équivalent de contraintes en logique du premier ordre. Ces instructions de trace sont insérées après chaque bloc d'affectation et chaque instruction conditionnelle du code, ce qui nous permet de récupérer les séquences de code activées à chaque exécution.

Le programme instrumenté (noté dans la figure 3.5 sous le nom `inst.c`) est une version du programme initial normalisé par CIL où l'on a ajouté des instructions de traçage.

Après la normalisation par CIL, l'instrumentation ajoute dans le code source normalisé des instructions de traçage pour chaque condition et bloc d'affectation (figure 3.8). Ces instructions contiennent les identifiants des conditions et blocs élémentaires d'affectations. Lors d'une exécution concrète, cet identifiant sera envoyé à l'étape de génération par les instructions de trace, ce qui permet au moteur d'exécution symbolique de récupérer le chemin exécuté. Ce dernier utilisera le chemin exécuté pour déterminer le prédicat du chemin du prochain cas de test. Pour l'exemple de la figure 3.1, le programme normalisé auquel on a ajouté des instructions de trace est présenté dans la figure 3.8. Une instruction

```
1 int cond(int x, int y) {
2   if (x > 0) {
3     pathcrawler_trace("+139");
4     if (x < 10) {
5       pathcrawler_trace("+140");
6       pathcrawler_trace(":141");
7       y++;
8     }
9     else {
10      pathcrawler_trace("-140");
11      goto _LAND;
12    }
13  }
14  else {
15    pathcrawler_trace("-139");
16    _LAND:
17    pathcrawler_trace(":144");
18    y--;
19  }
20  pathcrawler_trace(":145");
21  return(y);
22 }
```

FIGURE 3.8 – Version instrumentée du programme de la figure 3.1a

de trace est représentée par un appel à la fonction `pathcrawler_trace`. Cette fonction permet d'associer un identificateur à chaque condition et chaque bloc d'affectation. Par exemple, le bloc d'affectations qui contient l'affectation de la ligne 7 est identifié par le numéro `141`. Pour une même condition, nous notons par un "`+`" la branche "vrai" qui vérifie la condition, et par un "`-`" la branche "Faux". Le numéro `+140` identifie ainsi la branche *then* de condition à la ligne 4, et le numéro `-140` identifie la branche *else* de la même condition. Pendant l'exécution, la fonction `pathcrawler_trace` envoie ces numéros à la partie génération afin de lui transmettre la trace exécutée. Lorsque le générateur récupère la trace exécutée, il la simule en exécutant symboliquement les représentations symboliques de chaque numéro de la trace. Cela lui permet de déduire le prédicat satisfait par la trace et de générer le prochain cas de test. Considérons l'exemple du cas de test `cond(1,5)`. Ce test exécute le chemin suivant `(+139 +140 :141 :145)`. La représentation symbolique de cette trace est le prédicat de chemin suivant :

$$x_0 > 0 \wedge x_0 < 10 \wedge y_1 = y_0 + 1$$

où x_0 et y_0 qui représentent les valeurs initiales de x et y et x_i et y_i ($i \geq 1$) sont des variables introduites après chaque affectation pour exprimer x_i et y_i en fonction des valeurs précédentes dénotées x_{i-1} et y_{i-1} respectivement.

Préparation du lanceur

Le lanceur (nommé `lanceur.c` dans la figure 3.5) est un programme C permettant d'exécuter les cas de test. Une fois lancé, on peut lui transmettre (via un socket) les données de test à exécuter et on récupère par le même moyen le chemin parcouru. Il est ainsi possible d'exécuter séquentiellement plusieurs cas de test à l'aide du lanceur.

Un lanceur exécute dans une boucle infinie les actions suivantes :

1. il attend la réception des données de test (valeurs des variables d'entrée),
2. il initialise les variables selon les données de test reçues,
3. il exécute la version instrumentée de la fonction sous test qui retourne le chemin parcouru et le résultat obtenu (les sorties),
4. il appelle l'oracle fourni par l'utilisateur et retourne le verdict de test.

La figure 3.9 présente le lanceur d'un programme `test` qui prend en entrée deux tableaux d'entiers `T1` et `T2`, et deux entiers `l1` et `l2`. À la ligne 8, la fonction `pathcrawler_init_streams` initialise les canaux de communication entre le lanceur et le moteur d'exécution symbolique. La boucle des lignes 9 à 36 est la boucle principale du lanceur. Chaque itération exécute un cas de test différent. La fonction `pathcrawler_check_if_finished` (ligne 9) vérifie si le moteur d'exécution symbolique a signalé la fin de la génération. Dans ce cas le lanceur s'arrête. Les lignes 10 à 31 préparent le cas de test à exécuter à partir des données de test reçues. La fonction `pathcrawler_scan` permet de récupérer une donnée envoyée par le moteur d'exécution symbolique. À la ligne 10, elle permet de lire la taille à allouer pour le tableau `T1`. Si la taille `len` est égale à zéro, `T1` est initialisé à `NULL` (ligne 12). La fonction `pathcrawler_array_alloc` alloue la zone mémoire nécessaire pour `T1` (ligne 14). Ensuite on initialise le tableau `T1` à partir des informations reçues (lignes 15 à 17). De la même manière `T2` est aussi initialisé (lignes 20 à 29). Les entiers `l1` et `l2` sont initialisés dans les lignes 30 et 31. Ensuite la version instrumentée (décrite dans la section 3.3.3) de la fonction sous test est appelée (ligne 32). L'oracle est appelé à la ligne 33. Finalement la fonction `pathcrawler_free` libère les ressources allouées (lignes 34 et 35).

L'implémentation de ces fonctions dans leur version avant la thèse ne présente pas de difficulté particulière et ne sera pas détaillée ici.

3.3.4 Étape 2 : Génération des cas de tests

Nous sommes prêts à détailler la deuxième étape, présentée dans la figure 3.10, qui effectue la génération de tests. Lors de cette étape, PathCrawler utilise les fichiers intermédiaires générés lors de la première étape pour générer les cas de tests.

```
1 #include ‘‘inst.c’’
2 #include ‘‘lanceur.h’’
3 int main() {
4   int len;
5   int *T1, *T2;
6   int l1, l2;
7   int elem, i;
8   pathcrawler_init_streams();
9   while(!pathcrawler_check_if_finished()){

10    pathcrawler_scan(INT,&len);
11    if(len==0)
12      T1 = pathcrawler_null();
13    else{
14      T1 = pathcrawler_array_alloc(len, sizeof(int));
15      for(i = 0; i < len; i++){
16        pathcrawler_scan(INT, &elem);
17        T1[i] = elem;
18      }
19    }

20    pathcrawler_scan(INT,&len);
21    if(len==0)
22      T2 = pathcrawler_null();
23    else{
24      T2 = pathcrawler_array_alloc(len, sizeof(int));
25      for(i = 0; i < len; i++){
26        pathcrawler_scan(INT, &elem);
27        T2[i] = elem;
28      }
29    }

30    pathcrawler_scan(INT,&l1);
31    pathcrawler_scan(INT,&l2);

32    test(T1, T2, l1, l2);
33    pathcrawler_call_oracle();

34    pathcrawler_free(T1);
35    pathcrawler_free(T2);
36  }
37 }
```

FIGURE 3.9 – Exemple de lanceur pour un programme qui prend en entrée deux tableaux d’entiers et deux entiers

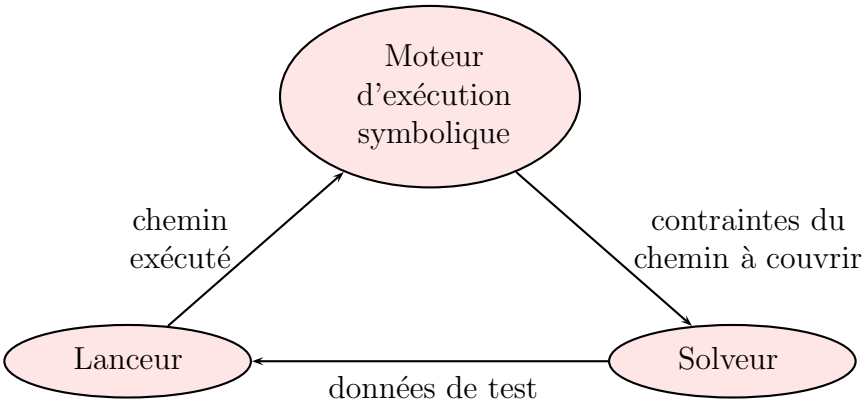


FIGURE 3.10 – Phase de génération de tests dans PATHCRAWLER

Avant de commencer la génération, on utilise `GCC` pour compiler le programme instrumenté avec le lanceur afin de pouvoir l'exécuter lors de l'exécution concrète.

Ensuite la génération effectue un parcours de tous les chemins en profondeur d'abord et génère un cas de test pour chaque chemin faisable. Cette étape de génération se décompose en quatre sous-parties :

1. Le parcours commence par le choix aléatoire d'un premier jeu de données en prenant en compte la précondition.

2. Le lanceur exécute concrètement le programme sous test sur le cas de test sélectionné et récupère le chemin exécuté par ce cas de test.

3. L'exécution symbolique consiste à calculer le prédicat du chemin à partir du dernier chemin obtenu. Ce chemin est utilisé pour trouver un nouveau chemin partiel non couvert avec la règle "*nie la dernière contrainte pas encore niée*". Ainsi, nous déterminons le domaine du prochain cas de test en réutilisant la structure du dernier prédicat calculé. L'exploration du graphe de flot de contrôle est donc contrôlée et optimisée, et elle se fait en profondeur d'abord.

   Un prédicat de chemin est la conjonction de contraintes sur les valeurs des variables d'entrée qui correspond à la conjonction des conditions de branchement dans lesquelles on a substitué les variables locales par l'expression de leurs valeurs en termes des valeurs des variables d'entrée. Pour calculer ce prédicat, il faut faire une projection des affectations vers les instructions conditionnelles afin d'exprimer l'expression conditionnelle par les valeurs des variables en entrée et ce, en fonction du chemin d'exécution suivi et en disposant d'une modélisation de la mémoire.

```
1 int tritype(int i, int j, int k){
2   int type = 0;
3   if (i == j)  // C1
4     type++;
5   if (i == k)  // C2
6     type++;
7   if (j == k)  // C3
8     type++;
9   return type;
10 }
```

FIGURE 3.11 – Programme `tritype`

4. La résolution et détermination du prochain cas de test. Le solveur résout les contraintes et donne une solution qui associe des valeurs aux entrées du programme. Cette solution constitue les données du cas de test activant ce prédicat de chemin.

Illustrons cette étape avec l'exemple du programme `tritype` présenté dans la figure 3.11. Le programme `tritype` calcule le type d'un triangle à partir des longueurs de ses segments. Le `type` est encodé comme suit :
- 0 : le triangle n'est ni isocèle ni équilatéral,
- 1 : le triangle est isocèle,
- 3 : le triangle est équilatéral.

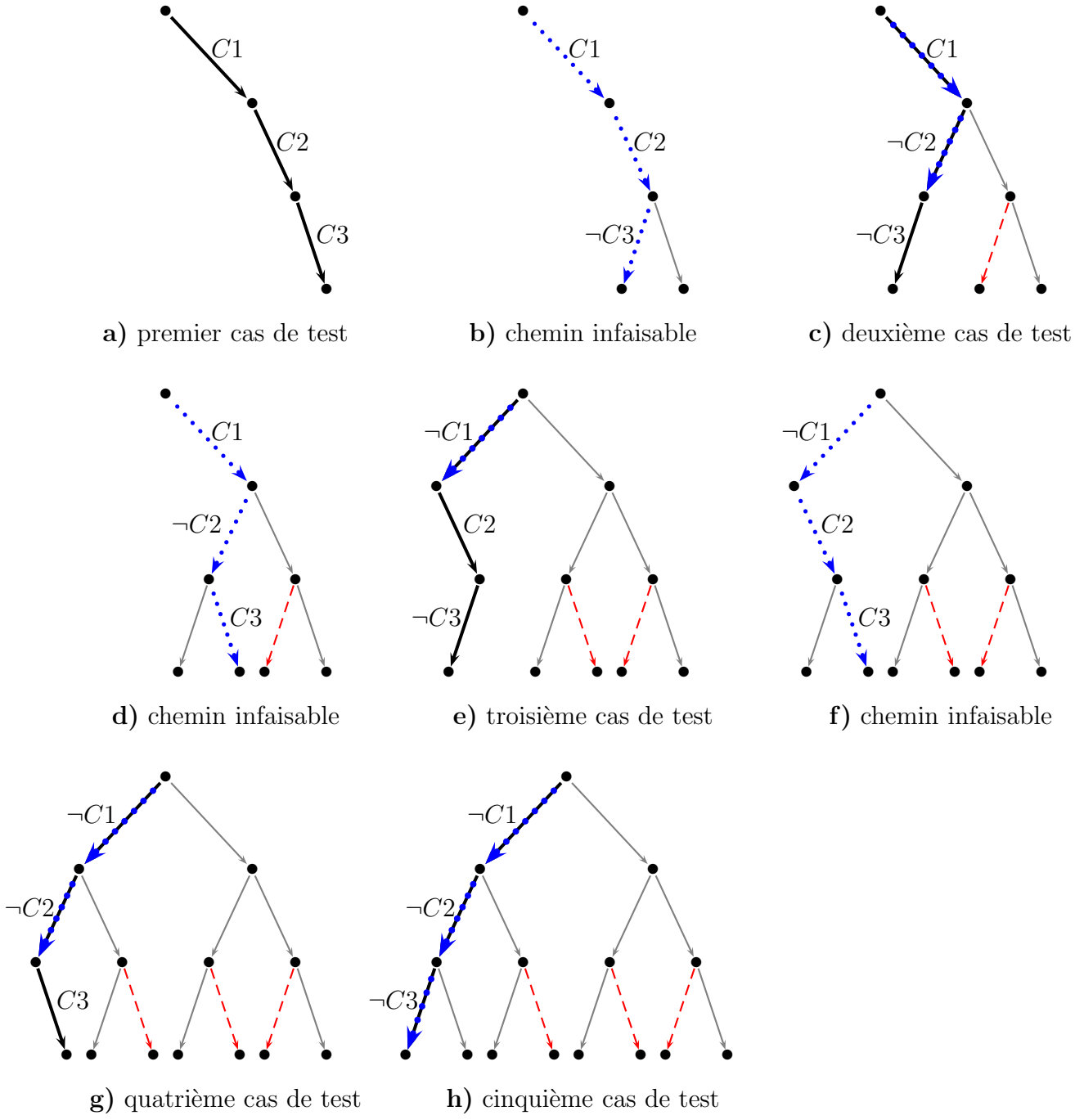
Les conditions aux lignes 3, 5 et 7 sont notées C1, C2 et C3 respectivement. Nous définissons la précondition de la fonction `tritype` comme suit :

$$1 \leq \texttt{i} \leq 10,\ 1 \leq \texttt{j} \leq 10,\ 1 \leq \texttt{k} \leq 10,\ \texttt{i} + \texttt{j} > \texttt{k},\ \texttt{i} + \texttt{k} > \texttt{j},\ \texttt{j} + \texttt{k} > \texttt{i}.$$

La figure 3.12 présente le déroulement de la génération de test sur cet exemple. Le prédicat de chemin est symbolisé par des flèches pointillées. Le cas de test courant est symbolisé par des flèches noires. Les chemins déjà explorés sont symbolisés par des flèches grises. Les chemins infaisables sont symbolisés par des flèches en tirets.

Pour le premier cas de test, il n'y a pas de contraintes autres que la précondition. Donc les valeurs sont choisies aléatoirement, par exemple `(i=1, j=1, k=1)`. Le chemin suivi par ce cas de test est $(C1 \wedge C2 \wedge C3)$. Il est représenté dans la figure 3.12a. Ce chemin est utilisé pour calculer le nouveau prédicat en niant sa dernière contrainte non encore niée, c'est-à-dire $C3$. On obtient le prédicat $(C1 \wedge C2 \wedge \neg C3)$ représenté dans la figure 3.12b.

Le système de contraintes $(C1 \wedge C2 \wedge \neg C3)$ étant insatisfaisable, un nouveau prédicat est calculé en utilisant la même règle. La dernière contrainte non encore niée dans le dernier chemin exécuté est $C2$. Le prédicat calculé est $(C1 \wedge \neg C2)$. Une solution à ce système de contraintes est le cas de test `(i=4, j=4, k=7)`. Le chemin emprunté est



a) premier cas de test **b)** chemin infaisable **c)** deuxième cas de test

d) chemin infaisable **e)** troisième cas de test **f)** chemin infaisable

g) quatrième cas de test **h)** cinquième cas de test

FIGURE 3.12 – Génération de tests pour le programme `tritype`

($C1 \wedge \neg C2 \wedge \neg C3$). Il est représenté dans la figure 3.12c.

Le chemin suivi par le cas de test précédent est analysé pour calculer le nouveau prédicat. Le prédicat obtenu est ($C1 \wedge \neg C2 \wedge C3$). Il est représenté dans la figure 3.12d. Étant insatisfaisable, on recalcule un nouveau prédicat. La dernière contrainte non encore niée dans le dernier chemin exécuté est $C1$. Le prédicat calculé en niant la dernière condition non encore niée est ($\neg C1$). Ce système de contraintes conduit au cas de test `(i=5, j=1, k=5)`. Le chemin suivi par ce cas de test est ($\neg C1 \wedge C2 \wedge \neg C3$). Il est représenté dans la figure 3.12e.

On reprend le chemin suivi par le test précédent et on calcule le nouveau prédicat ($\neg C1 \wedge C2 \wedge C3$). Aucun chemin suivant ce prédicat, représenté dans la figure 3.12f, n'est faisable puisque le système de contraintes ($\neg C1 \wedge C2 \wedge C3$) qui correspond au ($i \neq j; i = k; j = k$) est insatisfaisable. On calcule un nouveau prédicat en niant la dernière condition non encore niée. Le nouveau prédicat calculé est ($\neg C1 \wedge \neg C2$). Une solution au système de contraintes est `i=1, j=7, k=7`. Le prédicat ($\neg C1 \wedge \neg C2 \wedge C3$) et le chemin suivi par ce cas de test sont représentés dans la figure 3.12g.

On nie la dernière contrainte pour obtenir le nouveau prédicat ($\neg C1 \wedge \neg C2 \wedge \neg C3$). Une solution au système de contraintes est `i=10, j=5, k=6`. Le cas de test est représenté dans la figure 3.12h.

Tous les chemins d'exécution partiels ayant été explorés, le test est donc terminé pour cette fonction. Le graphe des chemins faisables (du graphe de flot de contrôle) a été découvert le long des différents cas de test comme le montre la figure 3.12. Les cinq chemins faisables de la fonction `tritype` ont bien été couverts et trois chemins partiels insatisfaisables ont été coupés de l'arbre de recherche.

Si un oracle est fourni par l'utilisateur, il sera exécuté après chaque cas de test pour produire le résultat attendu et le comparer au résultat obtenu. L'oracle est donc utilisé pour déterminer le verdict du cas de test exécuté. Ce verdict peut être "`success`" si les résultats sont conformes avec les résultats attendus. Dans le cas contraire, le verdict est "`failure`". Si l'oracle n'est pas fourni, le verdict du cas de test est "`unknown`".

La figure 3.13 présente les résultats d'une session de génération de test vus de l'interface graphique de PathCrawler.

L'ordre d'exploration des chemins est non déterministe et dépend du choix des cas de test.

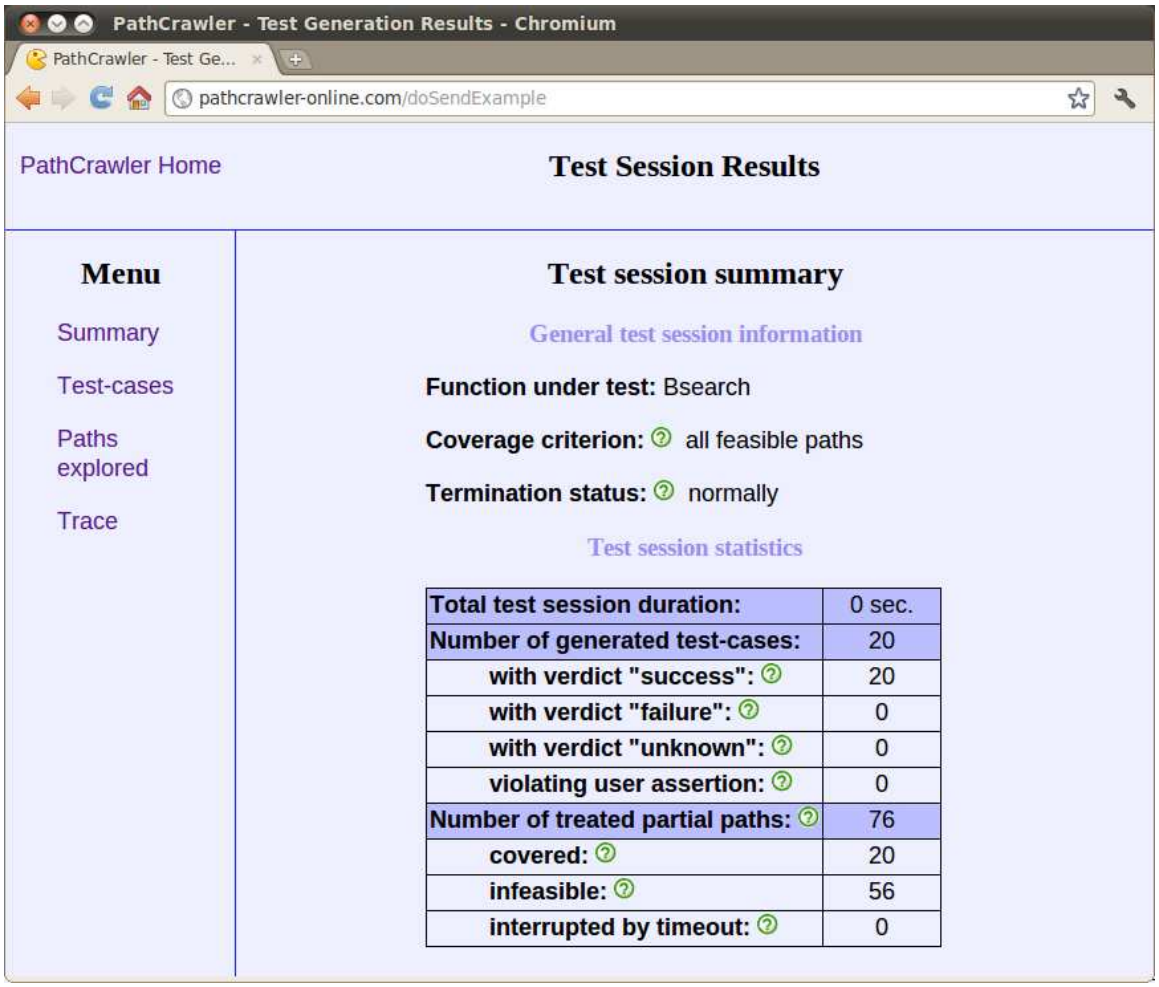


FIGURE 3.13 – Interface graphique de PATHCRAWLER

3.4 Frama-C

FRAMA-C [Fra11, CS09] (*Framework for modular analysis of C*) est une plateforme logicielle qui fournit un ensemble d'outils dédiés à l'analyse statique du code source d'un logiciel écrit en C. Il est développé en collaboration entre le CEA LIST et le projet Pro-Val de l'INRIA Saclay. Il est développé en OCAML et distribué en open source sur [Fra11].

FRAMA-C implémente un langage de spécification dédié, nommé ACSL (ANSI / ISO C Specification Language) [BFH+08] avec lequel on peut exprimer des spécifications fonctionnelles.

3.4.1 Architecture

La plateforme FRAMA-C dispose d'une architecture logicielle à greffons (*plug-ins*) similaire dans son esprit à celle de la plateforme Eclipse [Ecl11]. Elle intègre différents

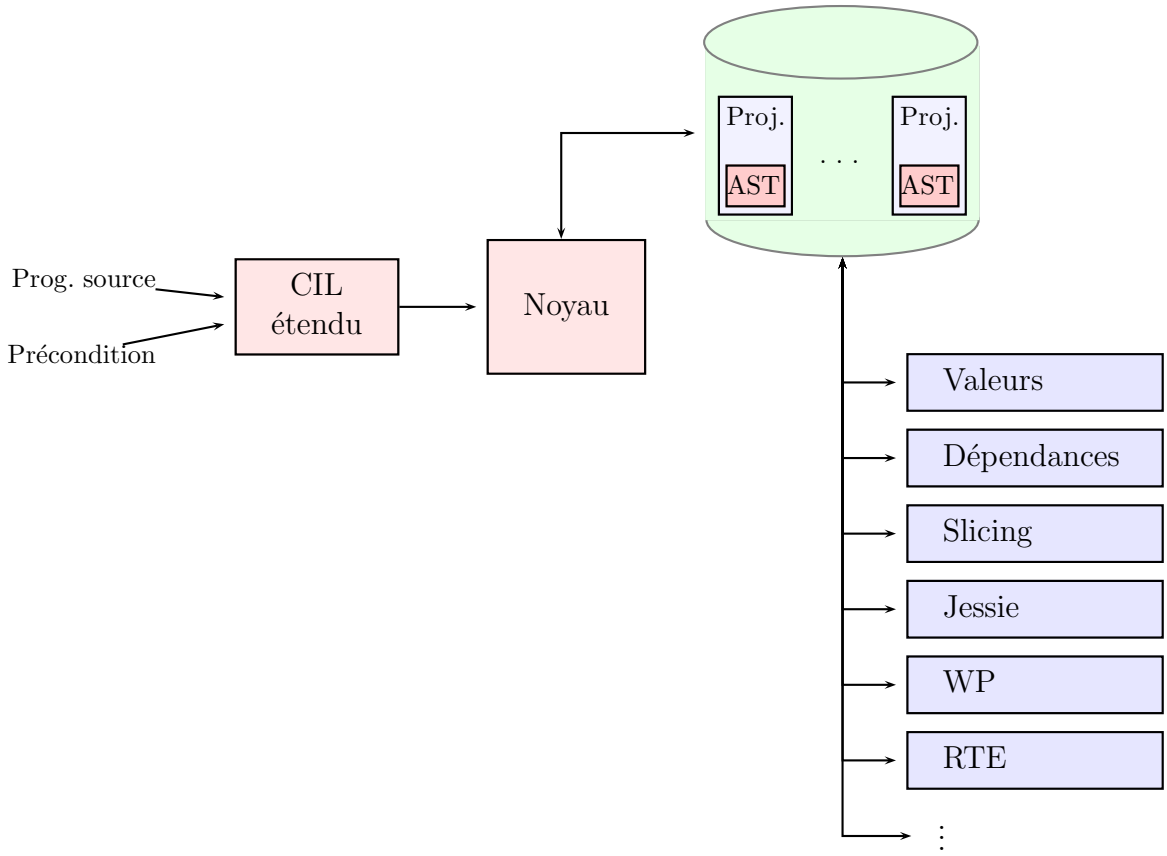


FIGURE 3.14 – Architecture de FRAMA-C

analyseurs comme autant de greffons. Ces derniers peuvent échanger des informations, ce qui permet de faire collaborer les analyseurs entre eux. Ainsi, par exemple, une analyse peut exploiter les résultats d'une autre.

L'architecture de FRAMA-C est présentée dans la figure 3.14. Son fonctionnement peut être découpé en trois phases :

1. Comme PATHCRAWLER, FRAMA-C utilise dans une première phase l'analyseur syntaxique CIL [CIL06] pour normaliser le code. Les transformations faites par CIL ont été présentées dans la section 3.2.1. FRAMA-C étend CIL avec le support du langage de spécification ACSL. Cette extension de CIL étend, par conséquent, l'interface d'utilisation ainsi que les types et les opérations afin de traiter correctement les programmes C annotés.

2. Le contrôle de l'analyse est effectué par le noyau. Le noyau de FRAMA-C centralise les informations et dirige l'analyse. De plus, le noyau fournit aux différents greffons un ensemble de fonctionnalités communes.

3. L'analyse est effectuée par les différents analyseurs qui sont les greffons centralisés autour du noyau dans Frama-C. Les greffons peuvent interagir les uns avec les autres. Chaque greffon inscrit ses services dans le noyau et les autres greffons peuvent ainsi exploiter ces services à travers des interfaces définies par le noyau. Parmi les services, on trouve l'analyse de valeurs, l'analyse de dépendances, la simplification syntaxique, le prouveur Jessie, le calcul de plus faible précondition, etc.

Frama-C est extensible et son approche collaborative facilite le développement de nouvelles analyses sous forme de greffons qui utilisent les services fournis d'une part par le noyau de Frama-C et d'autre part par les autres greffons existants.

Les analyseurs dans Frama-C peuvent également échanger des informations par le biais des annotations ACSL. Par exemple, un greffon qui a besoin de faire une hypothèse sur le comportement du programme peut exprimer cette hypothèse comme une propriété ACSL. Un autre greffon peut être utilisé ultérieurement pour prouver cette propriété.

Frama-C offre un outil de *slicing* qui permet de supprimer les parties non pertinentes d'un programme C selon un certain critère, tout en conservant un programme compilable. Les techniques de *slicing* sont présentées dans la section 2.4. Le *slicer* de Frama-C engendre un nouvel arbre de syntaxe abstraite plus petit que l'arbre initial. Cela permet à d'autres greffons d'analyser le programme simplifié directement, plutôt que le programme initial, sans produire son code C.

Frama-C offre la possibilité de travailler sur plusieurs arbres de syntaxe abstraite en parallèle. Chaque arbre est englobé dans ce qu'on appelle un **projet** Frama-C. Un **projet** Frama-C contient un arbre de syntaxe abstraite et les résultats des analyses sur cet arbre. Il est sauvegardé dans une base de données interne. Nous exploitons cette fonctionnalité pour analyser dans des projets différents les arbres de syntaxe astraite des programmes simplifiés produits par le *slicing*.

3.4.2 Langage de spécification ACSL

Le langage de spécification ACSL (ANSI / ISO C Specification Language) est un langage de spécification comportementale pour les programmes écrits en C. La conception de ACSL est inspirée de JML.

Les annotations ACSL se présentent dans le code sous forme de commentaires qui commencent par `//@` ou `/*@`. La figure 3.15 présente une partie de la spécification du programme `tritype` présenté dans la figure 3.11. Les lignes 1 à 6 présentent la précondition annotée avec le mot clé `requires`. La postcondition est indiquée grâce à l'annotation `ensures` (ligne 8). La ligne 7 précise que la fonction spécifiée n'a pas d'effet de bord, c'est-à-dire ne modifie aucune variable en dehors des variables locales.

```
1 /*@ requires 0 < i <= 100;
2     requires 0 < j <= 100;
3     requires 0 < k <= 100;
4     requires i + j > k;
5     requires i + k > j;
6     requires j + k > i;
7     assigns \nothing;
8     ensures \result == 0 || \result == 1 || \result == 3;
9  */
```

FIGURE 3.15 – Spécification du programme de la figure 3.11

Puisque ces annotations se situent dans des commentaires, elles ne gênent en rien la compilation du programme d'origine. Cependant, ACSL reste un langage formel et peut être utilisé par des programmes externes. Dans cette section, nous présentons une partie des annotations qui composent ce langage. Pour plus d'informations sur ACSL, voir le manuel de référence [BFH+08].

3.4.3 Les greffons

Les analyseurs suivants sont distribués avec FRAMA-C :

- L'analyse de valeurs [CCM09] utilise l'interprétation abstraite pour calculer une sur-approximation des domaines de variables à chaque instruction du programme.
- L'analyse de dépendances calcule les dépendances intra-procédurales entre les instructions dans un programme.
- Le *slicing* [Sli11] produit un programme simplifié par rapport à un critère donné.
- WP [CDA11] et Jessie [MM10] permettent de prouver que les fonctions C satisfont leurs spécifications. Ils implantent un calcul de plus faible précondition en se basant sur les annotations ACSL et le code C du programme. Ils génèrent des obligations de preuve pour chaque annotation qui sont ensuite vérifiées au sein d'assistants de preuve interactifs comme COQ, ISABELLE ou des prouveurs automatiques (ALT-ERGO, Z3, Simplify ...).
- RTE [Her11] génère des annotations qui spécifient l'absence d'erreur pour toutes les instructions potentiellement menaçantes qui peuvent provoquer une erreur à l'exécution ou un débordement arithmétique.

```
1 int div(int a) {
2   int x, y, res, tmp;
3   if(a){
4     x = 0;
5     y = 5;
6   }else{
7     x = 5;
8     y = 0;
9   }
10  tmp = x + y;
11  res = 10 / tmp;
12  return res;
13 }
```

FIGURE 3.16 – Exemple analysé avec l'analyse de valeurs

L'analyse de valeurs

L'analyse de valeurs [CCM09] de FRAMA-C calcule et sauvegarde des sur-ensembles des valeurs possibles des variables à chaque instruction du programme. Elle est basée sur l'interprétation abstraite. Elle commence à partir d'un point d'entrée (une fonction) spécifié par l'utilisateur dans le programme analysé, déplie les boucles et les appels de fonctions et calcule une sur-approximation des ensembles de valeurs possibles à chaque point du programme.

Pour l'exemple de la figure 3.16, l'analyse de valeur calcule les sur-ensembles des valeurs possibles l'exécution. À la ligne 3, l'analyse de valeurs calcule `a` $\in [--, --]$ ce qui signifie que le domaine de `a` varie de moins l'infini à plus l'infini. À la ligne 4, elle détecte qu'avant l'exécution de cette instruction `x` était non initialisée et qu'après l'exécution de cette instruction, `x` peut prendre uniquement la valeur zéro :

```
Before statement: x ∈ UNINITIALIZED

After statement: x ∈ {0}.
```

À la ligne 10, l'analyse de valeur calcule que `x` $\in$ `{0,5}` c'est-à-dire que `x` peut prendre l'une des deux valeurs 0 ou 5. Elle donne la même chose pour `y`. Pour `tmp`, elle détermine qu'avant l'exécution de cette instruction, `tmp` était non initialisée et qu'après l'exécution `tmp` $\in$ `{0,5,10}`. En réalité, après la ligne 10 `tmp` $\in$ `{5}` car `x` et `y` ne peuvent pas être nuls en même temps, mais cette analyse ne le détecte pas.

Les valeurs données par l'analyse de valeurs sont approximées mais correctes. Il se peut que certaines valeurs ne se produisent jamais à l'exécution mais il est garanti que toutes les valeurs possibles à l'exécution appartiennent aux ensembles calculés.

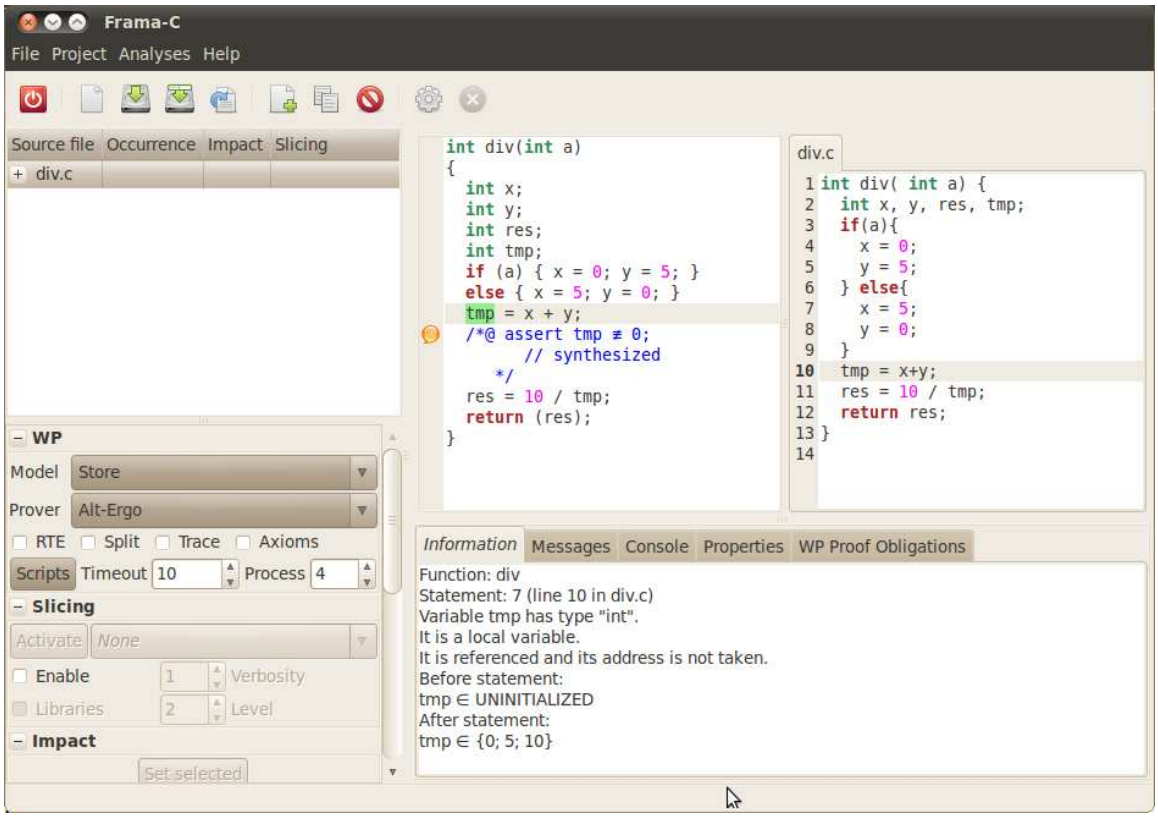


FIGURE 3.17 – Interface graphique de l'analyse de valeurs

La figure 3.17 présente l'interface graphique de l'analyse de valeurs dans l'environnement FRAMA-C. Un utilisateur peut sélectionner une expression dans le code pour voir la sur-approximation des ensembles de valeurs calculée pour cette expression.

Les ensembles calculés peuvent ainsi être utilisés, entre autres, pour détecter la présence de risque d'erreurs à l'exécution comme une division par zéro, un accès hors limite des indices d'un tableau, etc. Ils permettent aussi de prouver l'absence de ces dernières. L'analyse de valeurs va générer des alarmes pour les instructions pour lesquelles l'absence d'erreurs à l'exécution n'est pas assurée par les ensembles de valeurs calculés. Lorsque le risque d'une erreur d'exécution est exclu, aucune alarme n'est signalée.

Pour l'exemple de la figure 3.16, l'analyse de valeur émet une alarme de division par zéro à la ligne 11 :

`division by zero: assert tmp` $\neq$ `0.`

Une annotation est placée juste avant la ligne 11, avec le mot clé `assert`. Cette annotation contient la condition qui doit être respectée pour éviter une erreur à l'exécution (`tmp` $\neq$ `0`).

Cependant, les erreurs potentielles signalées par cette analyse peuvent s'avérer de fausses alarmes à cause de la sur-approximation comme l'alarme de division par zéro si-

gnalée pour le programme de la figure 3.16. En effet la valeur risquée (`tmp == 0`) ne se produit pas à l'exécution.

L'analyse de valeurs fournit des paramètres qui permettent d'améliorer la précision de l'analyse mais augmentent le temps de calcul et les ressources consommées. Notamment, il y a l'option `-slevel n`. Elle permet de mémoriser plusieurs (au plus n) états à chaque point de programme, en prenant en compte séparément les branches *then* et *else* d'une instruction conditionnelle, par exemple, ou en dépliant n itérations des boucles.

Pour l'exemple de la figure 3.16, analyser le programme avec l'option `-slevel 2` suffit pour prouver qu'il n'y a pas de risque de division par zéro à la ligne 11. En effet, l'analyse de valeurs calcule 2 états pour `x` et `y` à la ligne 11. Le premier état correspond à l'évaluation de la branche "vrai" de la conditionnelle à la ligne 3 (`x` ∈ `{0}`, `y` ∈ `{5}`). Le deuxième état correspond à l'évaluation de la branche "faux" (`x` ∈ `{5}`, `y` ∈ `{0}`). Dans les deux cas, la somme est différente de zéro.

Dans la pratique, d'une part il est difficile voire impossible de déterminer cette valeur n avec laquelle on a un bon compromis entre précision et temps de calcul, d'autre part, on ne sait pas à priori si les alarmes signalées sont de vraies bogues ou des fausses alarmes, donc on ne peut pas savoir jusqu'à combien il faut augmenter la précision.

L'analyse de valeurs est correcte dans le sens où elle ne reste pas silencieuse s'il y a un risque d'erreur à l'exécution. Elle peut prouver l'absence de bogues dans un programme, c'est-à-dire qu'il n'y a pas de risque d'erreur à l'exécution et que la spécification fonctionnelle ne peut pas être violée. Elle est automatique, étant donnés un code à analyser et éventuellement une précondition.

L'analyse de dépendances

Dans un programme, les instructions sont exécutées l'une après l'autre, et une instruction précédente peut avoir un impact sur les suivantes. L'analyse de dépendances calcule le graphe de dépendance de programme (*Program Dependence Graph* ou PDG) [FOW87] qui montre les relations de dépendance qui existent entre les instructions du programme.

Deux types de dépendances sont distingués : les dépendances de données et les dépendances de contrôle [BBD+10].

1. Une instruction i_2 est dépendante des données d'une autre instruction i_1 si i_1 modifie certaine variable x qui est utilisée par i_2 et s'il y a au moins un chemin dans le graphe de flot de contrôle de i_1 vers i_2 dans lequel x n'est pas modifiée entre i_1 et i_2.

2. Une instruction i_2 est dépendante du contrôle d'une autre instruction i_1 si i_1 est une instruction conditionnelle ou une instruction de boucle et l'évaluation de l'état de i_1 détermine si i_2 sera exécutée ou non. En d'autres termes, si i_1 est une instruction

```
1  int main(){
2    int a, c;
3    int b = 0;
4    a = 1;
5    for(c = 0; c <= 5; c++);
6    for(c = 0; c <= 5; c++){
7      a = 2;
8      b = a + 2;
9      b *= 2;
10   }
11   if(b)
12     b++;
13   for(c = 0; c <= 5; ){
14     a += 2;
15     a += 3;
16     goto H;
17     c++;
18   }
19   a++;
20   H:
21   if(a)
22     c++;
23   return a;
24 }
```

a) Programme initial

```
1  int main(){
2    int a, c;
3  --int b = 0;--
4    a = 1;
5  --for(c = 0; c <= 5; c++);--
6    for(c = 0; c <= 5; c++){
7      a = 2;
8  --  --b = a + 2;--
9  --  --b *= 2;--
10   }
11 --if(b)--
12 --  --b++;--
13   for(c = 0; c <= 5; ){
14     a += 2;
15     a += 3;
16     goto H;
17 --  --c++;--
18   }
19 --a++;--
20   H:
21 --if(a)--
22 --  --c++;--
23   return a;
24 }
```

b) Programme slicé par rapport à la variable `a` en ligne 23

FIGURE 3.18 – Réduction du code par le *slicing* de FRAMA-C

conditionnelle, alors i_2 appartient à la branche "vrai" ou la branche "faux" de i_1. Si i_1 est une boucle, alors i_2 appartient au corps de la boucle i_1.

Le *slicing*

Le *slicing* transforme un programme C en un programme C réduit, au regard de certains points de vue utilisateur, sur lequel il est possible d'effectuer des analyses afin de rendre les résultats plus pertinents. Les résultats peuvent être transposés sur le code initial.

Le programme slicé est un programme exécutable. Il est égal au programme initial dans lequel on a retiré certaines instructions. Il s'agit d'éliminer les instructions du code C initial qui ne participent pas au calcul ou à l'accessibilité du critère de *slicing*.

Un critère de *slicing* dans FRAMA-C peut être une variable locale ou globale en un point particulier du programme, une ligne de programme ou même une annotation ACSL.

Un critère de *slicing* peut également être composé de plusieurs de ces éléments.

Par exemple, pour le critère d'un ensemble de variables V en un point l du code donné, le *slicing* assure que si l'exécution sur un ensemble des données d'entrée E du code initial atteint l, alors on l'atteint également avec le code réduit, avec la même valeur pour ces variables. La réciproque est uniquement vraie pour les exécutions du code initial qui terminent.

La figure 3.18 illustre la réduction d'un programme par le *slicing* de Frama-C. Le programme initial est présenté dans la figure 3.18a. La figure 3.18b présente le programme slicé. Le critère du *slicing* est la variable `a` en ligne 23. Les lignes rayées sont les lignes supprimées par le *slicing*.

Ce greffon se base sur le greffon d'analyse de dépendances et effectue une autre analyse de dépendances interprocédurale permettant de prendre en compte les appels de fonction.

Jessie

Ce greffon permet de prouver que les fonctions C satisfont leurs spécifications exprimées en ACSL. Il est basé sur le calcul de plus faible précondition. Il génère des obligations de preuve pour chaque annotation. Ces obligations de preuve sont ensuite vérifiées à l'aide d'un assistant de preuve comme COQ, ISABELLE ou des prouveurs automatiques (ALT-ERGO, Z3, ...).

Ces preuves sont modulaires : les spécifications des fonctions appelées sont utilisées pour établir la preuve des appels de fonctions, sans analyser leurs codes. Jessie propose un modèle mémoire à séparation [Moy09]. L'utilisateur doit fournir la spécification de son programme et l'analyse nécessite une interaction entre l'analyseur et l'utilisateur.

WP

Ce greffon est basé sur le calcul de plus faible précondition comme Jessie. Mais, à la différence de Jessie, WP offre plusieurs modèles mémoire à différents niveaux d'abstraction. Chaque modèle est adapté à un type de problème. Les modèles mémoire peuvent être utilisés de manière modulaire.

Les assistants de preuves utilisés sont COQ, ISABELLE, ALT-ERGO, Z3, ...

3.5 Synthèse

Dans ce chapitre, nous avons présenté les deux outils PathCrawler et Frama-C. Le premier est un outil de génération de tests structurels des programmes écrits en C. Il

génère tous les chemins ou k-chemins du programme. FRAMA-C est une plate-forme pour l'analyse statique des programmes C. Nous avons présenté son architecture et quelques analyseurs fournis avec l'outil. Pour ce travail, nous en utilisons trois : l'analyse de valeurs, le *slicing* et l'analyse de dépendances.

Dans notre combinaison, nous utilisons l'analyse de valeurs pour prouver l'absence de certaines menaces et pour signaler les autres qui restent à vérifier. Nous utilisons l'analyse de dépendances entre les menaces pour déterminer les sous-ensembles de menaces dépendantes qui induisent un critère de *slicing*. Puis, nous utilisons ensuite le *slicing* pour simplifier le programme à analyser. Et enfin, nous utilisons PATHCRAWLER pour confirmer ou infirmer les menaces restantes.

Deuxième partie

Contributions

Chapitre 4

Méthode SANTE

Sommaire

4.1 Introduction

Dans ce chapitre, nous présentons de manière détaillée et formalisée la méthode SANTE dont l'originalité réside d'une part dans la combinaison de l'analyse de valeurs et l'analyse

dynamique et d'autre part dans l'ajout de l'étape du *slicing* qui englobe les différentes utilisations du *slicing* que nous avons introduites. Cette étape permet de surmonter les limites liées au manque de temps ou d'espace des approches ou combinaisons classiques. De plus, un bogue est signalé avec des informations plus précises sur l'erreur détectée, et illustré sur un programme plus simple, ce qui n'est pas le cas avec les approches classiques. Nous présentons les quatre utilisations du *slicing* et nous démontrons la correction de la méthode.

SANTE est principalement composée de trois étapes. L'étape de l'analyse de valeurs signale des menaces d'erreurs potentielles. Les résultats de cette étape guident l'étape de l'analyse dynamique. Le rôle de cette dernière est d'ajouter de la précision à la méthode en confirmant ou infirmant ces menaces. Une étape supplémentaire, utilisant le *slicing*, a été spécialement conçue et implémentée afin de lier les 2 analyses. Elle permet de réduire le programme afin de simplifier l'étape d'analyse dynamique. Nous expliquons les différentes étapes et les illustrons sur un exemple.

En quelques mots, l'analyse de valeurs est appelée en premier lieu et génère ainsi des alarmes quand elle ne peut pas garantir l'absence d'erreur. Ensuite, le programme est réduit par le *slicing*. Un ou plusieurs programmes simplifiés sont produits suivant les options fournies en entrée (les options d'utilisation du *slicing* dans SANTE sont présentées dans les parties 4.6 et 4.8). Puis l'analyse dynamique (génération de tests) est utilisée pour confirmer ou infirmer les alarmes dans le(s) programme(s) simplifié(s).

Ce chapitre est organisé comme suit. Nous commençons par la description générale de la méthode en partie 4.2. Dans la partie 4.2.1, nous présentons l'exemple `eurocheck` sur lequel la méthode sera illustrée tout au long de ce chapitre. Les parties 4.3, 4.4 et 4.5 sont dévolues aux différentes étapes de la méthode. Les options de base de la méthode sont présentées dans la partie 4.6. La partie 4.7 étudie les dépendances entre les alarmes sur lesquelles se basent les options avancées (partie 4.8). La partie 4.9 prouve d'une part la correction de la méthode et explique d'autre part son incomplétude.

4.2 Description générale de la méthode

Cette méthode peut être représentée sommairement par le schéma de la figure 4.1. Elle comporte trois étapes principales. L'étape d'**analyse statique** génère un ensemble d'alarmes. L'étape **Slicing** produit, un ou plusieurs programmes simplifiés dont chacun préserve un ensemble non vide d'alarmes. À la dernière étape l'**analyse dynamique** analyse les programmes simplifiés et produit le diagnostic.

L'étape d'analyse statique utilise l'**analyse de valeurs** de FRAMA-C [CCM09] qui va générer l'ensemble des alarmes $A = alarms(p)$ pour les instructions d'un programme p, pour lesquelles l'absence d'erreur à l'exécution n'est pas assurée par analyse statique étant donné une précondition. Par exemple, pour la ligne 2 de l'exemple de la figure 4.2, le gref-

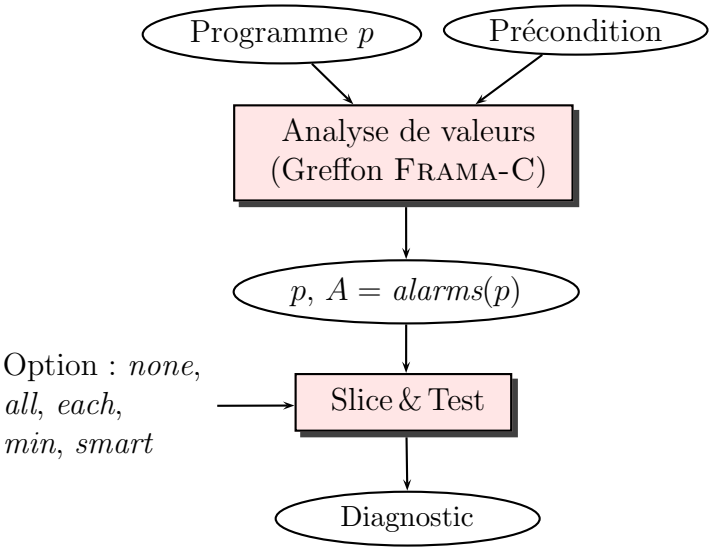


Figure 4.1 – La méthode SANTE

fon émet *"Alarme : z peut être 0 !"* et retourne la condition caractérisant un état d'erreur *(z == 0)* si 0 est contenu dans l'ensemble de valeurs calculé pour z avant cette instruction.

Ensuite, ces alarmes sont transférées à la fonction **Slice & Test** qui englobe les étapes de *slicing* et d'analyse dynamique. La fonction Slice & Test utilise ces alarmes pour simplifier le programme sous test par le *slicing*. L'étape **Slice** produit un ou plusieurs programmes simplifiés dont chacun préserve un ensemble non vide d'alarmes. L'union de toutes les alarmes préservées par tous les programmes simplifiés est égal à toutes les alarmes générées par l'analyse de valeurs. L'utilisation du *slicing* se fait en fonction d'options fournies en entrée. Dans SANTE, on a implémenté cinq options : *none*, *all*, *each*, *min* et *smart*, qui correspondent à différentes heuristiques pouvant être utilisées lors de la simplification. Les options sont présentées dans les parties 4.6 et 4.8.

La dernière étape **Test** consiste à analyser dynamiquement le(s) programme(s) simplifié(s). L'analyse dynamique utilise les alarmes signalées par l'analyse de valeurs pour guider la génération de tests. Elle essaie de confirmer ces alarmes en activant des erreurs sur certains cas de test lors de son parcours de tous les chemins ou de tous les k-chemins. Nous appelons cette technique *"alarm-guided test generation"* ou "génération de tests guidée par les alarmes".

Un diagnostic est ainsi produit pour chaque alarme. Ce diagnostic peut prendre l'une des valeurs suivantes :

- safe si l'état d'erreur est prouvé inatteignable,
- bug si l'analyse dynamique trouve des entrées menant à l'erreur et
- unknown quand ce n'est pas possible de tirer des conclusions sur l'état d'erreur, c'est-à-dire, quand la génération est arrêtée en raison de dépassement du temps limite ou quand un critère de couverture partielle est utilisé.

```
0 int Division(int x, int z){
1   int y;
2   y = x/z;
3   return y;
4 }
```

FIGURE 4.2 – Exemple Division

Cette approche passe par la génération automatique des branches d'erreurs correspondant à la transition symbolique à couvrir (l'instruction menaçante) avec des conditions sur les variables caractérisant un état d'erreur. Dans l'exemple de la figure 4.2, si l'alarme est émise, l'instruction menaçante `y=x/z;` sera remplacée par

```
if(z==0)
  error();
else
  y=x/z;
```

4.2.1 Exemple d'illustration

Nous illustrons chaque étape de la méthode sur une version simplifiée de l'exemple `eurocheck-0.1.0` présentée dans la figure 4.3. Le code source du programme initial est libre et peut être téléchargé sur [Rut11].

Etant donné une chaîne de caractères `str` représentant le numéro de série d'un billet de banque en euros, cette fonction vérifie si le numéro de série est valide ou non. Ce numéro devrait commencer par une lettre, suivie de plusieurs chiffres. La lettre identifie le pays émetteur (Ex : U pour la France). Le programme calcule le code de contrôle (*checksum*) à partir des chiffres et le compare avec les identificateurs des pays.

Nous définissons la précondition de la fonction `eurocheck` comme suit :

`str` est un tableau de caractères de taille ≥ 0.

4.3 L'analyse de valeurs

Un programme p muni de sa précondition est présenté à l'entrée de cette phase. Des instructions de ce programme peuvent être associées à des opérations risquées comme une division ou un accès à un tableau. L'ensemble de ces instructions constitue la liste exhaustive des menaces potentielles dans ce programme. Le rôle de l'analyse statique est de déterminer l'absence d'erreur pour certaines d'entre elles. SANTE commence par appliquer l'analyse de valeurs (voir figure 4.1) pour éliminer le plus grand nombre possible

```
0  int eurocheck(char *str){
1    unsigned char sum;
2    char c[9][3]={"ZQ","YP","XO",
3     "WN","VM","UL","TK","SJ","RI"};
4    unsigned char checksum[12];
5    int i = 0, len = 0;
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32; //capitalize
8    if(str[0]<'I' || str[0]>'Z')
9      return 2; //invalid char
10   if(strlen(str) != 12)
11     return 3; //wrong length
12   len = strlen(str);
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15     if(str[i]<48 || str[i]>57)
16       return 4; //not a digit
17     checksum[i] = str[i]-48}
18   sum=0;
19   for(i=(len-1);i>=1;i--)
20     sum+=checksum[i];
21   while(sum>9)
22     sum=((sum/10)+(sum%10));
23   for(i=0;i<9;i++)
24     if(checksum[0]==c[i][0] || checksum[0]==c[i][1])
25       break;
26   if(sum!=i)
27     return 5; //wrong checksum
28   return 0;} //OK
```

FIGURE 4.3 – Programme `eurocheck`

de menaces potentielles.

Dans FRAMA-C, chaque instruction reçoit une étiquette ou "label" implicite et unique, même quand il n'y a pas d'étiquette explicite dans le code C. Nous identifions l'instruction par cette étiquette unique. Nous désignons par l_a (le label de) l'instruction menaçante de l'alarme a et, dans nos exemples, par k (le label de) l'instruction à la ligne k. Une alarme a est donc vue comme un couple (l_a, c_a) contenant le label de l'instruction menaçante l_a et la condition d'erreur potentielle c_a de l'alarme a. Cette condition contient des variables référencées (lues) à l'instruction menaçante l_a. L'erreur signalée par l'alarme a se produit à l_a si et seulement si la condition c_a est vraie avant l'exécution de l_a. Par exemple, l'alarme d'accès en dehors des limites du tableau `checksum` à la ligne 17 de la figure 4.3 est le couple $(17, i < 0 \vee i >= 12)$. Cette alarme est représentée dans le programme de la figure 4.4 par une clause `assert` $\neg C_a$ à la ligne 17_0.

```
0  int eurocheck(char *str){
1    unsigned char sum;
2    char c[9][3]={"ZQ","YP","XO",
3     "WN","VM","UL","TK","SJ","RI"};
4    unsigned char checksum[12];
5    int i = 0, len = 0;
6₀   //@ assert(\valid(str+0));
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32;
8    if(str[0]<'I' || str[0]>'Z')
9      return 2;
10   if(strlen(str) != 12)
11     return 3;
12   len = strlen(str);
13₀ //@ assert(\valid(str+i));
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15₀    //@ assert(\valid(str+i));
15     if(str[i]<48 || str[i]>57)
16       return 4;
17₀    //@ assert((0<=i) ∧ (i<12));
17     checksum[i] = str[i]-48}
18   sum=0;
19   for(i=(len-1);i>=1;i--)
20₀    //@ assert((0<=i) ∧ (i<12));
20     sum+=checksum[i];
21   while(sum>9)
22     sum=((sum/10)+(sum%10));
23   for(i=0;i<9;i++)
24     if(checksum[0]==c[i][0] || checksum[0]==c[i][1])
25       break;
26   if(sum!=i)
27     return 5;
28   return 0;}
```

FIGURE 4.4 – Programme `eurocheck` avec les alarmes

L'analyse de valeurs [CCM09] de FRAMA-C va calculer et sauvegarder les sur-ensembles des valeurs possibles des variables à chaque instruction du programme. Ces ensembles peuvent ainsi être utilisés, entre autres, pour exclure la possibilité d'une erreur à l'exécution. L'analyse de valeurs va générer des alarmes pour les instructions pour lesquelles l'absence d'erreur à l'exécution n'est pas assurée par les ensembles de valeurs calculés. Par exemple, pour l'instruction à la ligne 13 de la figure 4.3, l'analyse de valeurs ne peut pas garantir que `str+i` fait référence à un emplacement mémoire valide, donc elle émet une alarme indiquant que `str+i` pourrait être invalide. Pour l'instruction à la ligne 20,

l'analyse de valeurs ne peut pas exclure le risque d'un accès hors limite dans les valeurs calculées pour l'indice `i` et elle émet une alarme indiquant que `checksum+i` pourrait être invalide.

Pour le programme p de la figure 4.3, l'analyse de valeurs renvoie l'ensemble $A = \mathit{alarms}(p)$ contenant cinq alarmes :

$$\begin{gathered}(\,6,\ 0 \geq \texttt{length(str)}\,),\\ (\,13,\ \texttt{i} < 0 \vee \texttt{i} \geq \texttt{length(str)}\,),\\ (\,15,\ \texttt{i} < 0 \vee \texttt{i} \geq \texttt{length(str)}\,),\\ (\,17,\ \texttt{i} < 0 \vee \texttt{i} \geq \texttt{12}\,),\\ (\,20,\ \texttt{i} < 0 \vee \texttt{i} \geq \texttt{12}\,).\end{gathered}$$

À la ligne 6, nous lisons le premier caractère `str[0]`. Cette alarme est une erreur parce que `str` peut être vide. A la ligne 13, l'analyse de valeurs signale que `str[i]` peut être un accès hors-limites. Cette alarme est une fausse alarme parce que si la longueur de `str` n'est pas égale à 12, l'exécution du programme s'arrête à la ligne 11 et retourne `3` ce qui signifie que la chaîne n'a pas la bonne longueur et l'exécution n'atteindra jamais la ligne 13. A la ligne 15, l'alarme signalée est également une fausse alarme. Il en est de même pour les alarmes à la ligne 17 et à la ligne 20. Ces imprécisions sont dues au fait que l'analyse de valeurs ne déroule pas toutes les itérations des boucles, mais calcule des sur-approximations.

Techniquement, l'analyse de valeurs de Frama-C marque chaque instruction menaçante de p par une annotation placée juste avant, en utilisant le mot-clé `assert`. Ces assertions représentent les alarmes. Le programme de la figure 4.3 dans lequel on a ajouté les alarmes est présenté dans la figure 4.4. Par exemple, à la ligne 15, les valeurs calculées pour `i` contiennent des valeurs supérieures à la longueur de la chaîne `str` et l'annotation :

```
//@ assert(\valid(str+i));
```

est ajoutée juste avant la ligne 15 (voir la ligne 15_0 dans la figure 4.4) pour signaler que l'accès au tableau `str[i]` peut être hors-limite. L'annotation donne aussi la condition qui doit être vérifiée pour éviter la menace, c'est-à-dire, la négation ($\neg c_a$) de la condition d'erreur c_a. Pour les pointeurs, le passage de `\valid(str+i)` à une condition d'erreur c_a sur l'indice `i` n'est pas toujours immédiat. Dans la partie 5.3 du chapitre 5, nous allons détailler ce point en expliquant comment nous implémentons la traduction d'une assertion en une condition d'erreur exécutable.

L'analyse de valeurs est correcte dans le sens où, si elle n'émet pas d'alarme pour une menace potentielle, alors l'erreur ne peut pas se produire pour cette menace. Mais en général, elle n'est pas très précise, et certaines alarmes signalées peuvent être de fausses alarmes. Les propriétés satisfaites par l'analyse de valeurs utilisée dans sante sont définies dans la proposition 4.3.1.

Proposition 4.3.1 (Complétude de l'analyse de valeurs)
(a) *Prouver l'absence d'erreur par l'analyse de valeurs est correct : si l'analyse de valeurs n'émet pas d'alarme pour une menace potentielle, une erreur ne peut pas se produire pour*

```
 0 int eurocheck(char *str){
 1   unsigned char sum;
 2   char c[9][3]={"ZQ","YP","XO",
 3     "WN","VM","UL","TK","SJ","RI"};
 4   unsigned char checksum[12];
 5   int i = 0, len = 0;
 6₀  //@ assert(\valid(str+0));
 6   if(str[0]>=97 && str[0]<=122)
 7     str[0]-=32;
 8   if(str[0]<'I' || str[0]>'Z')
 9     return 2;
10   if(strlen(str) != 12)
11     return 3;
12   len = strlen(str);
13₀ //@ assert(\valid(str+i));
13   checksum[i]=str[i];
14   for(i=1;i<len;i++){
15₀    //@ assert(\valid(str+i));
15     if(str[i]<48 || str[i]>57)
16       return 4;
17₀    //@ assert(0 <= i ∧ i < 12);
17     checksum[i] = str[i]-48}
18   sum=0;
19   for(i=(len-1);i>=1;i--)
20₀    //@ assert(0 <= i ∧ i < 12);
20     sum+=checksum[i];}
21  while(sum>9)
22    sum=((sum/10)+(sum%10));
23  for(i=0;i<9;i++)
24    if(checksum[0]==c[i][0] || checksum[0]==c[i][1])
25      break;
26  if(sum!=i)
27    return 5;
28  return 0;}
```

FIGURE 4.5 – *Slice* du programme de la figure 4.4 sur toutes les alarmes – Le programme slicé est égal au programme initial dans lequel on a retiré les parties rayées

cette alarme.
(b) *Prouver l'absence d'erreur par l'analyse de valeurs n'est en général pas précis : certaines alarmes signalées peuvent être des fausses alarmes.*

```
0  void eurocheck(char *str){
5    int i, len;
6₀   //@ assert \valid(str+0);
6    if(str[0]>=97 && str[0]<=122)
7      str[0]-=32;
8    if(str[0]<'I' || str[0]>'Z')
9      return;
10   if(strlen(str) != 12)
11     return;
12   len = strlen(str);
14   for(i=1;i<len;i++){
15₀    //@ assert \valid(str+i);
15     if(str[i]<48 || str[i]>57)
16       return;}}
```

FIGURE 4.6 – Le programme de la figure 4.4 simplifié par rapport à l'alarme à la ligne 15

4.4 Le *slicing*

Le *slicing* [Wei81] est une technique de transformation de programmes qui permet d'extraire un sous-programme exécutable, appelé *slice*, à partir d'un programme plus grand. Une *slice* a le même comportement que le programme original par rapport au critère du *slicing*. Le *slicing* de FRAMA-C est basé sur l'analyse de dépendance qui comprend le calcul du graphe de dépendance de programme (PDG) [FOW87] montrant les relations de dépendance entre les instructions du programme, et une analyse de dépendance interprocédurale permettant de traiter les appels de fonction. Le critère classique du *slicing* est une paire composée d'une instruction et d'un ensemble de variables du programme. Le *slicing* de FRAMA-C accepte d'autres critères de *slicing*, par exemple un ensemble d'instructions.

Notons $\rightsquigarrow$, la fermeture réflexive et transitive de la relation de dépendance de données ou de contrôle. En d'autres termes, $l_1 \rightsquigarrow l_2$ si $l_1 = l_2$, ou si l'exécution de l_2 dépend de l'exécution de l_1 (directement ou via des états intermédiaires). Cette relation n'est pas symétrique : si $l_1 \rightsquigarrow l_2$, cela n'implique pas que $l_2 \rightsquigarrow l_1$. Par exemple pour les lignes 7 et 8 de la figure 4.3, nous avons $7 \rightsquigarrow 8$, mais nous n'avons pas $8 \rightsquigarrow 7$. Mais nous pouvons avoir des dépendances mutuelles $l_1 \rightsquigarrow l_2$ et $l_2 \rightsquigarrow l_1$ entre instructions. C'est le cas pour les deux lignes 21 et 22 de la figure 4.3.

Nous désignons par $\mathit{labels}(p)$ l'ensemble des étiquettes des instructions d'un programme p. Soit L un sous-ensemble de $\mathit{labels}(p)$. Essentiellement, dans les techniques de *slicing* basé sur les dépendances (voir par exemple [RY88, Sec. 2.2] et [BBD+10, définition 4.6]), la *slice* de p par rapport à L, notée $\mathit{slice}(p, L)$ ou p_L, est définie comme le sous-programme de p contenant les instructions suivantes :

$$\mathit{labels}(p_L) = \{\, l \in \mathit{labels}(p) \mid \exists\, l' \in L,\, l \rightsquigarrow l' \,\}. \tag{4.1}$$

Pour un singleton $L = \{l\}$, la *slice* p_L est également notée p_l. Comme $\rightsquigarrow$ est réflexive, $L \subseteq \mathit{labels}(p_L)$ et $l \in \mathit{labels}(p_l)$.

La *slice* d'un programme par rapport à une alarme a est définie comme étant la *slice* par rapport à l'instruction menaçante l_a de a, et on écrit p_a au lieu de p_{l_a}. De même, la *slice* d'un programme par rapport à un ensemble d'alarmes A est définie comme la *slice* par rapport à l'ensemble des instructions menaçantes de A, et on écrit p_A au lieu de $\mathit{slice}(p, \{l_a \mid a \in A\})$.

L'étape du *slicing* produit une ou plusieurs versions simplifiées de p, chacune d'elles contenant un sous-ensemble d'alarmes qui peuvent être déclenchées. Nous avons étudié et implémenté cinq utilisations du *slicing* qui correspondent aux cinq options présentes dans SANTE.

1. ***none*** : Le programme p est directement analysé par analyse dynamique sans aucune simplification par le *slicing*.

2. ***all*** : Le *slicing* est appliqué une fois et le critère de simplification est l'ensemble de toutes les alarmes de p. Nous obtenons une version simplifiée contenant les mêmes menaces que le programme original p. Ensuite, l'analyse dynamique est appliquée à cette version simplifiée.

3. ***each*** : Soit n le nombre d'alarmes dans p. Le *slicing* est effectué n fois, une fois par rapport à chaque alarme a_i, produisant n programmes simplifiés $p_1, p_2, \ldots, p_n$. Ensuite, l'analyse dynamique est appelée n fois pour analyser les n programmes simplifiés.

4. ***min*** : On sélectionne des sous-ensembles d'alarmes $A_1, A_2, \ldots, A_k$ de A tels que $A_1 \cup A_2 \cup \ldots \cup A_k = A$ avec $k \leq n$. L'opération de *slicing* est effectuée k fois pour $A_1, A_2, \ldots, A_k$. Ainsi k programmes simplifiés $p_{A_1}, p_{A_2}, \ldots, p_{A_k}$ sont produits. Ensuite, l'analyse dynamique est appelée k fois pour analyser les k programmes résultants p_{A_i}.

5. ***smart*** : Cette option applique l'option *min* itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire.

Les trois premières options (*none*, *all* et *each*) sont les options de base. Elles sont présentées en détail dans la partie 4.6. *min* et *smart* sont des options plus avancées, qui se basent sur une analyse de dépendances entre les alarmes présentée en partie 4.7. Les options *min* et *smart* sont détaillées en partie 4.8.

Pour le programme en figure 4.4, le programme simplifié qui préserve toutes les alarmes est présenté dans la figure 4.5. Afin de faciliter la comparaison avec la version initiale,

les parties rayées montrent le code enlevé par le *slicing* et les parties non rayées correspondent au code du programme simplifié. Un autre programme simplifié est donné par la figure 4.6. Il présente la version simplifiée par rapport à l'alarme à la ligne 15. On notera que le code restant a été légèrement simplifié par suppression des initialisations de `i` et de `len` et par suppression des valeurs retournées. On constate que la *slice* par rapport à une seule alarme est plus simple que celle relative à toutes les alarmes.

Certaines propriétés du *slicing* utilisent une sémantique à base des trajectoires finies que nous présentons brièvement dans la partie 4.4.1, en s'inspirant de [BBD+10].

4.4.1 Sémantique à base de trajectoires

Un *état* est une fonction totale qui associe une valeur à chaque variable. Une valeur non définie est notée $\perp$. La *restriction d'un état à un ensemble de variables* V, notée par $s{\downarrow}V$, est définie par :

$$(s{\downarrow}V)x = \begin{cases} s(x) & \text{si } x \in V, \\ \perp & \text{sinon.} \end{cases}$$

Une séquence non vide $T = (l_1, s_1)(l_2, s_2)\dots(l_k, s_k)\dots$ de couples *(instruction, état)* est appelée *trajectoire de* p si, à partir d'un état d'entrée s_1, le programme p exécute la séquence d'instructions $l_1, l_2, \dots, l_k, \dots$ où l_i est l'étiquette de la i-ème instruction exécutée et s_i est l'état avant l'exécution de l_i. On dit aussi que T est la *trajectoire du programme* p *à partir de l'état d'entrée* s_1.

On dit que U est une *trajectoire partielle de* T si U est finie et T commence avec U. En d'autres termes, les trajectoires partielles de T sont les préfixes finis de T. Ils ont la forme suivante :

$$(l_1, s_1)(l_2, s_2)\dots(l_m, s_m),\ m \geq 1.$$

Soit $M \subseteq \mathit{labels}(p)$ un ensemble d'étiquettes de p. La *restriction d'un élément de trajectoire* (l, s) *sur* M est définie comme suit :

$$(l, s){\downarrow}M = \begin{cases} (l, s{\downarrow}\mathrm{ref}(l)) & \text{si } l \in M, \\ \varepsilon & \text{sinon,} \end{cases}$$

où $\mathrm{ref}(l)$ est l'ensemble des variables lues à l'instruction l et ε est la séquence vide.

Pour une trajectoire $T = (l_1, s_1)\dots(l_k, s_k)\dots$, la *projection de* T *sur* M est définie par :

$$\mathrm{Proj}_M(T) = (l_1, s_1){\downarrow}M\ \dots\ (l_k, s_k){\downarrow}M\ \dots$$

Le *slicing* utilisé dans SANTE doit vérifier les propriétés suivantes :

```
0  void eurocheck(char *str){
5    int i, len;
6₀   //@ assert \valid(str+0);
6₁   if(0 >= length(str))
6₂     error();
6₃   else
6      if(str[0]>=97 && str[0]<=122)
7        str[0]-=32;
8    if(str[0]<'I' || str[0]>'Z')
9      return;
10   if(strlen(str) != 12)
11     return;
12   len = strlen(str);
14   for(i=1;i<len;i++){
15₀    //@ assert \valid(str+i);
15₁    if(i >= length(str))
15₂      error();
15₃    else
15       if(str[i]<48 || str[i]>57)
16         return;}}
```

FIGURE 4.7 – Le programme simplifié par rapport à l'alarme à la ligne 15 auquel ont été ajoutées les branches d'erreurs

PROPOSITION 4.4.1 (PROPRIÉTÉS DU SLICING)
Soit L un ensemble d'instructions dans p, p_L la slice de p par rapport à l'ensemble d'instructions L et s_0 un état d'entrée pour p. Soit T et T' respectivement les trajectoires de p et p_L exécutées à partir de l'état d'entrée s_0.
(a) *labels(p_L) est inclus dans labels(p).*
(b) *Si U est une trajectoire partielle de T, alors il existe une trajectoire partielle U' de T' telle que les projections de U et U' sur L sont égales :* $\text{Proj}_L(U) = \text{Proj}_L(U')$.

4.5 L'analyse dynamique

Le *slicing* est suivi par une dernière étape d'analyse dynamique qui sera appliquée à tous les programmes simplifiés. L'analyse dynamique essaie d'activer chaque alarme. Pour ce faire, elle essaie de couvrir les chemins d'exécution dans lesquels les erreurs sont déclenchées. Cette étape produit pour chaque alarme un diagnostic : safe, bug ou unknown.

On définit la fonction de l'analyse dynamique DA comme suit :

DÉFINITION 1 (ANALYSE DYNAMIQUE)
Soit p un programme et A un ensemble d'alarmes présentes dans p. La fonction d'analyse dynamique DA appliquée à p calcule une fonction de diagnostic sur A qui associe à chaque alarme $a \in A$ l'un des résultats suivants :

1. *safe, ce qui signifie qu'il n'y a pas d'erreur dans p pour a,*
2. *un couple* (bug, s)*, ce qui signifie qu'une erreur pour a se produit dans p exécuté à partir de l'état d'entrée s,*
3. *unknown, ce qui signifie que nous ne savons pas s'il y a une erreur dans p pour a.*

L'alarme est dite classée si son diagnostic est bug ou safe. En particulier, la fonction d'analyse dynamique retourne le diagnostic (bug, s) pour l'alarme $a = (l_a, c_a)$ si et seulement s'il existe une trajectoire $T = (l_1, s_1), \ldots, (l_k, s_k), \ldots$ telle que $s_1 = s$, $l_k = l_a$ et la condition d'erreur c_a est satisfaite dans l'état s_k, donc l'erreur signalée par a se produit vraiment à l'exécution de l_a à partir de l'état s_k et à l'exécution du programme p à partir de s_1. Le diagnostic safe est retourné si tous les chemins ont été explorés (donc s'ils sont tous finis), sans atteindre l'état d'erreur.

L'implémentation de la fonction d'analyse dynamique DA doit garantir que, lorsque la génération de test se termine en ayant couvert tous les chemins exécutables et qu'elle n'a pas couvert certains chemins du programme, c'est qu'il n'existe aucun état d'entrée qui permette de couvrir ces chemins. Nous avons choisi d'utiliser l'approche *concolique*, aussi nommée *exécution dynamique symbolique*, pour la génération de tests couvrant tous les chemins [Kos10]. L'implémentation utilisée doit rapporter tout parcours incomplet, par exemple, avec un parcours des chemins arrêté par un temps limite avant de couvrir certains chemins, ou une résolution de contraintes arrêtée par un temps limite avant de générer un cas de test pour un chemin, ou une imprécision due à des fonctionnalités non supportées du langage C, ou une simplification non équivalente des contraintes de chemins (ce qui est fait de façon silencieuse par certains outils pour améliorer leurs performances). L'outil PathCrawler utilisé dans notre implémentation vérifie ces conditions et rapporte tout parcours incomplet des chemins du programme.

Cette étape d'analyse dynamique est décomposée en deux sous-étapes. La première sous-étape, présentée dans la partie 4.5.1, adapte l'approche *concolique* pour obtenir notre fonction d'analyse dynamique DA. La deuxième sous-étape, présentée dans la partie 4.5.2, est la génération de tests.

4.5.1 Ajout des branches d'erreur

Afin de forcer la génération de tests à activer les erreurs possibles sur chaque chemin du programme p, nous ajoutons des *branches d'erreur* dans le code source de p de la façon suivante. Pour chaque alarme $a = (l_a, c_a)$, l'instruction menaçante l_a, notée ci-dessous

```
  threatStatement;
```

est remplacée par l'instruction conditionnelle suivante :

```
if( c_a )
  error();
else
  threatStatement;
```

La génération de tests est ensuite exécutée pour le programme C résultant, noté p'. Nous appelons cette technique *alarm-guided test generation* ou "génération de tests guidée par les alarmes". Pour l'exemple de la figure 4.3, le programme slicé sur l'alarme à la ligne 15, auquel ont été ajoutées les branches d'erreurs, est présenté dans la figure 4.7. Les trois lignes 15_1, 15_2 et 15_3 ont été ajoutées.

Si la condition d'erreur est vérifiée dans p', alors une erreur à l'exécution peut se produire, et la fonction `error()` signale l'erreur et arrête l'exécution du cas de test en cours. S'il n'y a pas d'erreur à l'exécution, cette dernière se poursuit normalement et p' se comporte exactement comme p.

4.5.2 Génération de tests

Les techniques d'évaluation symbolique à l'aide de la programmation par contraintes [Kos10] sont utilisées pour concrétiser la menace et générer un cas de test.

Dans notre implémentation, nous utilisons l'outil PathCrawler [BDHTH+09] dont la méthode est similaire à la génération de *test concolique* [SMA05], aussi appelé *l'exécution dynamique symbolique*. L'utilisateur fournit le code source C de la fonction sous test. Le générateur explore les chemins du programme dans une recherche en profondeur d'abord en utilisant l'exécution symbolique et concrète.

La transformation de p en p' ajoute de nouvelles branches qui séparent les états d'erreur des états sans erreur. L'algorithme de PathCrawler va automatiquement essayer de couvrir les états d'erreur. Pour une alarme a, PathCrawler peut la confirmer comme un bug quand il trouve un état d'entrée et un chemin d'erreur menant à l'état d'erreur. PathCrawler peut également prouver que l'alarme est safe lorsque la génération de tests pour tous les chemins de p' se termine sans l'activation de la menace correspondante. Lorsque la génération de tests pour les chemins de p' n'est pas complète, aucune alarme n'est classée safe. Dans PathCrawler, cela se produit s'il y a trop de chemins (la génération ne termine pas avant un temps limite), ou si la résolution des contraintes pour un chemin ne termine pas (avant un temps limite), ou si le programme sous test ne termine pas (avant un temps limite) sur un test, ou lorsqu'on utilise un critère de couverture partielle (k-chemins), ou lorsque le programme contient des fonctionnalités non supportées du langage C. Enfin, toutes les alarmes qui ne sont pas classées comme bug ou safe demeurent unknown.

Dans la *slice* présentée sur la figure 4.7, la génération de tous les chemins se termine sans couvrir l'instruction à la ligne 15_2, donc cette instruction est inatteignable et il n'y a aucun état d'entrée qui peut déclencher l'alarme correspondante. Par conséquent, cette alarme est classée safe.

L'analyse dynamique utilisée dans SANTE satisfait les propriétés suivantes :

Proposition 4.5.1 (Sûreté de l'analyse dynamique)
(a) *La classification d'une alarme par l'analyse dynamique est correcte, c'est-à-dire que si l'analyse dynamique classe une alarme comme* bug *ou* safe*, alors elle l'est vraiment.*
(b) *La classification d'une alarme par l'analyse dynamique peut en général être incomplète, c'est-à-dire que certaines alarmes peuvent rester* unknown*.*

4.5.3 Diagnostic partiel à partir d'une analyse dynamique sur une *slice*

Dans notre méthode, nous allons appliquer DA à plusieurs versions simplifiés $p_1, p_2, \ldots,$ p_m de p. Cependant, le diagnostic produit par DA sur une *slice* p_j $(1 \leq j \leq m)$ pour une alarme de p ne donne pas toujours le statut réel de cette menace dans le programme initial p. Dans certains cas, une alarme peut être une erreur confirmée sur la version simplifiée p_j, alors que dans le programme initial son statut peut rester inconnu ou même safe. En revanche, si le diagnostic produit par DA sur p_j pour une alarme est safe, alors cette alarme ne représente aucun risque d'erreur dans le programme initial p (nous allons le prouver dans la partie 4.9). Pour mieux expliquer ce phénomène d'apparition d'erreur dans les *slices*, considérons deux situations que nous illustrons par des exemples. Soit $A_j \subset A$ le sous-ensemble d'alarmes de A préservées dans la *slice* p_j.

Une erreur pour une alarme $a \in A_j$ peut se produire dans la *slice* p_j sur un état d'entrée s sans se produire dans p si l'exécution de p sur s n'atteint pas l'instruction menaçante correspondante l_a. Cela peut être dû à une partie avant l_a qui boucle et ne termine pas dans le programme p et qui n'est plus présente dans la version simplifiée. Cette situation peut être illustrée par l'exemple suivant :

```
0 int f1(int x, int y){
1   int z;
2   g(x,y);  // Does not terminate
3   z=x/y;   // Alarm: y may be 0
4   return z;
5 }
```

où on suppose que la variable `y` peut être nulle, la fonction `g(x,y)` ne termine pas et l'appel `g(x,y);` n'est pas une dépendance pour la ligne 3. Dans ce cas, la *slice* par rapport à la ligne 3 est la suivante :

```
0 int f1(int x, int y){
1   int z;
3   z=x/y;   // Alarm: y may be 0
4   return z;
5 }.
```

Elle contient en effet une erreur pour l'alarme de la ligne 3 (pouvant être confirmée pour les entrées `(x,y)=(1,0)` par exemple) qui ne se produit jamais dans le programme initial où l'exécution boucle infiniment sans jamais arriver à la ligne 3.

Une autre cause possible d'apparition d'erreur dans les *slices* peut être liée au phénomène des erreurs cachées. Une erreur pour une alarme $a \in A_j$ peut se produire dans la *slice* p_j sur un état d'entrée s sans se produire dans p si l'exécution de p sur s produit une autre erreur avant l'instruction menaçante l_a. Par exemple, considérons le programme suivant :

```
0 void f2(int x, int y, int c){
1   int a,b;
2   a=x/c;   // Alarm: c may be 0
3   printf("␣new␣x=%d␣",a);
4   b=y/c;   // Alarm: c may be 0
5   printf("␣new␣y=%d␣",b);
6   return;
7 }
```

où la variable `c` peut être nulle. Dans ce cas, la *slice* par rapport à la ligne 4

```
0 void f2(int x, int y, int c){
1   int b;
4   b=y/c;   // Alarm: c may be 0
6   return;
7 }
```

provoque une erreur pour l'alarme de la ligne 4 (par exemple, sur `(x,y,c)=(5,10,0)`) alors que l'exécution du programme initial ne produit pas d'erreur pour cette ligne car l'exécution s'arrête suite à une erreur de division par zéro à la ligne 2. Nous voyons que l'erreur de la ligne 2 cache l'erreur de la ligne 4 dans le programme initial. Une erreur cachée peut se produire dans le programme simplifié à cause de la simplification.

Dans la méthode SANTE, pour éliminer toute incertitude sur le statut d'une menace a pour laquelle DA sur une *slice* p_j a retourné (bug, s), nous proposons de confirmer ce statut par l'exécution du programme initial p sur s. L'exécution concrète de p est immédiate, et permet de répondre précisément si l'état d'entrée s provoque une erreur sur p pour la menace a ou non. Si l'erreur pour la menace a se produit dans p, nous pouvons confirmer le diagnostic (bug, s) pour a dans p. Si cette erreur ne se produit pas dans p, alors le statut de l'alarme a dans p reste inconnu tant que cette alarme n'est pas confirmée par l'exécution de p sur un autre état d'entrée.

Dans ce cas, si lors de l'exécution de p sur s, une erreur pour une autre alarme $a' \in A$ se produit dans p, cette information sera conservée et le diagnostic (bug, s) sera émis pour a' dans p. Nous pouvons émettre un avertissement pour attirer l'attention sur l'alarme a, qui peut être une erreur cachée, mais nous ne pouvons pas conclure sur le statut exact de a. Pour simplifier la présentation, ce type d'avertissement n'est pas représenté dans la présentation formelle de la méthode et le diagnostic.

Si l'exécution de p sur s ne termine pas (dans un délai fixé), ce cas de non terminaison doit être rapporté et analysé pour détecter une éventuelle erreur menant à la non terminaison. La détection de la non terminaison étant un problème indécidable dont l'étude ne

fait pas partie des objectifs de cette thèse, nous nous limitons à signaler les éventuels cas détectés de non terminaison (dans un délai fixé) et n'incluons pas ces avertissements dans la formalisation de la méthode et le diagnostic.

En pratique, dans nos expérimentations présentées dans cette thèse, l'exécution du programme initial a toujours confirmé le statut d'erreur rapporté par DA sur une *slice*.

Nous pouvons maintenant donner la définition formelle du diagnostic $Diagnostic^j$ pour les alarmes de p à partir des résultats obtenus par DA sur une *slice* p_j $(1 \leq j \leq m)$.

Définition 2 (Diagnostic à partir d'une slice)
Le diagnostic $Diagnostic^j$ pour les alarmes de p à partir d'une analyse dynamique sur la slice p_j est défini pour tout $a \in A$ comme suit.
1. *$Diagnostic^j(a) =$ safe si $a \in A_j$ et DA sur la slice p_j a classé a comme* safe.
2. *$Diagnostic^j(a) = (\mathsf{bug}, s)$ si $a \in A_j$, DA sur la slice p_j a retourné (bug, s) pour a et l'exécution de p sur s confirme l'erreur pour la menace a dans p.*
3. *$Diagnostic^j(a) = (\mathsf{bug}, s)$ si l'exécution de p sur s (lors d'une tentative de confirmer le statut d'erreur d'une autre menace $a' \in A_j$ pour laquelle DA sur la slice p_j a retourné (bug, s)) provoque l'erreur pour la menace a dans p avant d'arriver à a'.*
4. *$Diagnostic^j(a) =$ unknown dans tous les autres cas.*

4.5.4 Fusion de diagnostics et construction du diagnostic final

Lorsque DA est appliqué à plusieurs versions simplifiées $p_1, p_2, \ldots, p_m$ de p et qu'ainsi plusieurs diagnostics pour les alarmes de p sont obtenus, nous avons besoin de cumuler ces diagnostics pour obtenir un diagnostic final.

Montrons d'abord que les diagnostics $Diagnostic^j$ et $Diagnostic^k$ (si $j \neq k$) obtenus à partir des résultats de DA sur deux *slices* p_j et p_k ne peuvent pas produire deux résultats contradictoires safe et bug. Si l'un des diagnostics est bug, alors une erreur se produit réellement pour a dans p (voir la définition 2). On prouve dans la partie 4.9 que si DA sur une *slice* produit le diagnostic safe pour une alarme a, alors il n'y a aucun risque d'erreur pour a sur p. Par conséquent, les diagnostics $Diagnostic^j$ et $Diagnostic^k$ ne peuvent pas produire en même temps deux diagnostics contradictoires safe et bug pour une même alarme a.

Cette propriété de non contradiction entre les diagnostics nous permet d'introduire l'opération de fusion de diagnostics par la définition suivante.

Définition 3 (Fusion de Diagnostics)
Soit p un programme et A un ensemble d'alarmes présentes dans p. Soient $Diagnostic'$ et $Diagnostic''$ deux diagnostics pour l'ensemble A d'alarmes de p. La fusion des diagnostics

Diagnostic′ et Diagnostic″, notée Diagnostic′⊎Diagnostic″, est définie comme suit pour toute alarme a de A :
1. *Diagnostic′ ⊎ Diagnostic″(a) =* safe *si Diagnostic′(a) =* safe *ou Diagnostic″(a) =* safe*.*
2. *Diagnostic′⊎Diagnostic″(a) = (*bug*, s) si Diagnostic′(a) = (*bug*, s) ou Diagnostic″(a) = (*bug*, s).*
3. *Diagnostic′ ⊎ Diagnostic″(a) =* unknown *dans tous les autres cas.*

Pour la présentation des algorithmes de la méthode SANTE ci-dessous, il sera pratique d'avoir une représentation des diagnostics en termes d'ensembles d'alarmes de chaque statut. Soient $Diagnostic'$ et $Diagnostic''$ deux diagnostics pour l'ensemble A des alarmes de p, et soit $Diagnostic = Diagnostic' \uplus Diagnostic''$ leur fusion. Soient S', B', U' les ensembles d'alarmes pour lesquels le diagnostic $Diagnostic'$ retourne, respectivement, safe, bug et unknown. Soient S'', B'', U'' les ensembles d'alarmes pour lesquels le diagnostic $Diagnostic''$ retourne, respectivement, safe, bug et unknown. Alors les ensembles S, B, U d'alarmes pour lesquels le diagnostic $Diagnostic$ retourne, respectivement, safe, bug et unknown, peuvent être calculés de la manière suivante :

$$\begin{aligned} S &= S' \cup S'', \\ B &= B' \cup B'', \\ U &= A - (S \cup B). \end{aligned}$$

On utilisera également dans ce cas la notation $(S, B, U) = (S', B', U') \uplus (S'', B'', U'')$, ou même $(S, B, U) = (S', B', U') \uplus Diagnostic''$. Il est clair que la fusion de diagnostics est une opération commutative et associative.

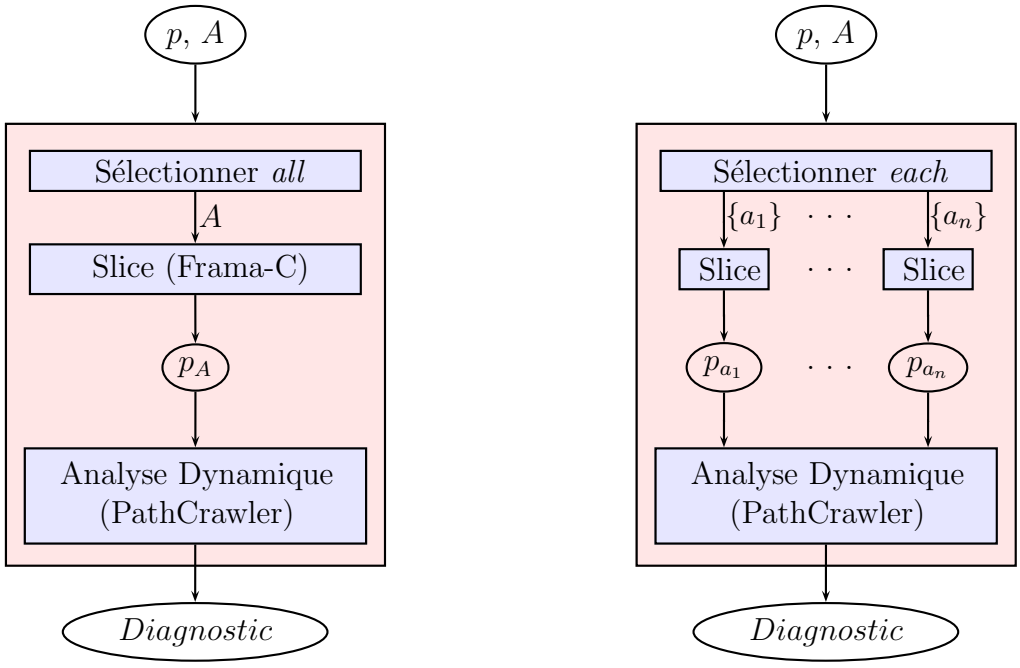
Dans la méthode SANTE, le diagnostic final sera construit par fusion des diagnostics obtenus pour les alarmes de p à partir de l'application de DA sur les versions simplifiées $p_1, p_2, \ldots, p_m$ de p.

4.6 Slice & Test : options de base

Dans cette partie, nous présentons les options de base de la fonction Slice & Test : *none*, *all* et *each*. A désigne l'ensemble des alarmes d'un programme p.

4.6.1 Option *none*

Le programme p est directement analysé par analyse dynamique sans aucune simplification par le *slicing*. Le principal inconvénient de cette option est que l'analyse dynamique sur un grand programme peut prendre beaucoup de temps ou ne pas terminer en laissant beaucoup d'alarmes non classées, c'est-à-dire avec le statut unknown. Nous avons proposé



a) Option *all* **b)** Option *each*

FIGURE 4.8 – Slice & Test : options de base

ce fonctionnement dans l'article [CKGJ10b].

Avec cette option, SANTE prend 25 secondes environ pour confirmer l'alarme à la ligne 6 comme `bug` et classer les quatre autres alarmes comme `safe`, dans l'exemple `eurocheck` de la figure 4.3.

4.6.2 Option *all*

Avec cette option, illustrée par la figure 4.8a, le *slicing* est appliqué une seule fois et le critère de simplification est l'ensemble A contenant toutes les alarmes de p (l'ensemble des instructions menaçantes contenant ces alarmes). On obtient une version simplifiée p_A contenant les mêmes menaces que le programme original p. Ensuite, l'analyse dynamique est appliquée à p_A.

Les avantages de cette option sont clairs. Nous obtenons un programme plus simple p_A contenant les mêmes menaces que le programme initial p. L'analyse dynamique fonctionne plus rapidement que pour p, parce qu'elle est appliquée à sa version simplifiée p_A. Pour l'exemple de la figure 4.3, p_A ne contient que 19 lignes (voir figure 4.5) au lieu de 29 (voir figure 4.3). Avec cette option, les cinq alarmes sont classées en environ 7 secondes.

Cependant, comme le programme p_A contient toutes les alarmes présentes dans A, l'analyse dynamique peut manquer de temps ou d'espace parce que certaines alarmes peuvent être complexes ou difficiles à analyser. Dans ce cas, les alarmes qui sont plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes, et finalement un grand nombre d'alarmes peuvent rester non classées. Pour remédier à cet inconvénient, nous introduisons l'option *each*.

4.6.3 Option *each*

Soit $n \geq 1$ le nombre d'alarmes dans p et $A = \{a_1, a_2, \ldots, a_n\}$ l'ensemble des alarmes dans p. Avec cette option, illustrée dans la figure 4.8b, le *slicing* est effectué n fois, une fois par rapport à chaque alarme a_i, produisant un programme simplifié p_{a_i} pour chaque alarme a_i ($1 \leq i \leq n$). Ensuite, l'analyse dynamique est appelée n fois pour analyser les n programmes résultants p_{a_i}.

L'avantage de cette option est la production, pour chaque alarme a_i, du programme simplifié minimal p_{a_i} qui préserve l'instruction menaçante de l'alarme a_i. Par conséquent, chaque alarme est analysée séparément (autant que possible) par analyse dynamique, donc aucune alarme ne reste arbitrairement pénalisée par une autre. L'analyse dynamique de chaque *slice* p_{a_i} est plus rapide que celle de p et a plus de chance de classer a_i dans un délai fini.

Pour l'exemple en cours (figure 4.3), on obtient cinq programmes simplifiés dont les tailles varient de 3 à 16 lignes. La figure 4.6 montre l'exemple du programme simplifié par rapport à la menace à la ligne 15. Pour chacun des cinq programmes simplifiés, la génération de tests dure moins de 6 secondes. Le temps complet requis par SANTE pour analyser les cinq *slices* et classer les cinq alarmes est d'environ 13 secondes.

Parmi les inconvénients de cette option, premièrement le *slicing* est exécuté n fois et l'analyse dynamique est exécutée pour n programmes. En outre, une *slice* peut en inclure une autre ou être identique à une autre. Dans ces cas, l'analyse dynamique pour certains des p_{a_i} est une perte de temps. Cela est dû aux dépendances (mutuelles) entre les menaces. Par exemple, dans le programme de la figure 4.3, l'instruction à la ligne 15 dépend de l'instruction à la ligne 6. Donc l_6 est présent dans le programme simplifié par rapport à l'alarme à la ligne 15 (voir figure 4.6). L'analyse dynamique pour p_{15_0} peut classer l'alarme 6_0. Dans ce cas on n'a pas besoin d'analyser p_{6_0}.

Dans la partie 4.7, nous étudions les dépendances entre les menaces et nous les utilisons dans la partie 4.8 pour introduire les options avancées de Slice & Test.

4.7 Dépendances entre alarmes

Les résultats de cette partie sont applicables à l'ensemble de toutes les alarmes de p et à n'importe quel sous-ensemble. Soit $A \subseteq \mathit{alarms}(p)$ un ensemble d'alarmes de p. Rappelons que l'alarme a est considérée comme un couple (l_a, c_a) contenant l'instruction menaçante l_a et la condition d'erreur potentielle c_a de a. On dit qu'une alarme $a' \in A$ dépend d'une autre alarme $a \in A$ et on écrit $a \rightsquigarrow a'$ si $l_a \rightsquigarrow l_{a'}$, c'est-à-dire si l'instruction menaçante $l_{a'}$ de a' dépend de l'instruction menaçante l_a de a.

On suppose que chaque instruction est une instruction menaçante pour au plus une alarme. Donc, pour simplifier la notation, lorsque la condition d'erreur n'est pas référencée, on confond une alarme $a \in A$ avec l'instruction menaçante qui lui correspond l_a. De même un ensemble d'alarmes est considéré comme l'ensemble des étiquettes des instructions. Par exemple, lorsque $a = (l, c)$ est une alarme dans A, on peut utiliser une des deux notations : $a \in A$ et $l \in A$, sans risque de confusion.

Quand deux alarmes a et $a' \in A$ sont indépendantes, le *slicing* par rapport à a élimine a' dans le programme simplifié. Mais, dans la plupart des cas, les alarmes ne sont pas toutes indépendantes, et une alarme $a \in A$ peut dépendre d'une autre alarme $a' \in A$. Deux alarmes peuvent également être mutuellement dépendantes ($a_1 \rightsquigarrow a_2$ et $a_2 \rightsquigarrow a_1$) comme dans l'exemple suivant :

```
1 for(i=0; i<3; i++) {
2   y = x/z - 1; // Alarme: z peut être 0
3   z = x/y - 2; // Alarme: y peut être 0
4  }
```

où les lignes 2 et 3 dépendent l'une de l'autre et les valeurs $(\mathtt{x}, \mathtt{y}, \mathtt{z}) = (2, 1, 1)$ et $(2, 1, 2)$ provoquent des bugs pour les deux alarmes. Suite à la définition d'une *slice* (voir définition 4.1), pour toute alarme $a \in A$, l'ensemble d'étiquettes $\mathit{labels}(p_a) \cap A$ est l'ensemble des instructions menaçantes de A qui survivent dans p_a. Comme $a \in \mathit{labels}(p_a)$, on a :

$$A = \bigcup_{a \in A} \mathit{labels}(p_a) \cap A.$$

Quand certaines alarmes sont dépendantes, on peut recouvrir A en ne prenant qu'une partie des *slices* p_a dans la partie droite de cette égalité. Cette idée nous conduit à la notion de couverture des alarmes par *slicing*.

Définition 4 (Couverture des alarmes par slicing)
Soit $A' \subseteq A$. On dit que le sous-ensemble A' définit une couverture des alarmes par slicing de A si la famille $(\mathit{labels}(p_a) \cap A \,|\, a \in A')$ est une couverture de A, c'est-à-dire si

$$A = \bigcup_{a \in A'} \mathit{labels}(p_a) \cap A.$$

On appelle une telle famille $(\mathit{labels}(p_a) \cap A \,|\, a \in A')$ la couverture des alarmes par slicing de A définie par A'.

Dans une telle couverture, chaque sous-ensemble $labels(p_a) \cap A$ est non vide. Les ensembles $labels(p_a) \cap A$ de cette famille seront appelés **ensembles couvrants**.

Définition 5 (Alarme Finale)
Soit e une alarme dans l'ensemble des alarmes A. On dit que e est une alarme finale de $(A, \rightsquigarrow)$ *si, pour tout* $s \in A$ *telle que* $e \rightsquigarrow s$*, on a* $s \rightsquigarrow e$*. En d'autres termes, une alarme finale n'a pas de dépendances sortantes autres que celles qui sont mutuelles.*

Comme A est fini, il est facile de voir que, pour tout $a \in A$, il y a une alarme finale $e \in A$ qui dépend d'elle, c'est-à-dire $a \rightsquigarrow e$. On note $ends(A)$ l'ensemble des alarmes finales de A.

On considère la relation de dépendance mutuelle $a \sim a'$ qu'on définit comme $a \rightsquigarrow a'$ et $a' \rightsquigarrow a$. C'est une relation d'équivalence dans A dont les classes d'équivalence sont les sous-ensembles maximaux d'alarmes interdépendantes dans A. On note $\overline{a}$ la classe d'équivalence de a. Le lemme 4.7.1(a) montre que si une classe d'équivalence contient une alarme finale $e \in A$, alors tous ses éléments sont des alarmes finales. On note $ends(A/\sim)$ l'ensemble des classes d'équivalence des alarmes finales. D'autres propriétés des alarmes finales et des *slices* sont données dans le lemme 4.7.1 :

Lemme 4.7.1
Soit $A \subseteq alarms(p)$ *un ensemble d'alarmes d'un programme p.*
(a) *Si e est une alarme finale dans A alors chaque élément de sa classe d'équivalence* $\overline{e}$ *est aussi une alarme finale dans A.*
(b) *Si* $L \subseteq A$ *et e est une alarme finale dans A qui survit dans la slice* p_L*, alors il existe un* $l \in L$ *tel que* $e \sim l$*.*
(c) *Si* $a \in A$ *et e est une alarme finale dans A qui survit dans la slice* p_a*, alors* $e \sim a$*.*
(d) *Si* $a \sim a'$ *sont deux alarmes équivalentes dans A, alors* $p_a = p_{a'}$*.*
(e) *Si* $a \in A$ *et* $A' = labels(p_a) \cap A$*, alors* $p_a = p_{A'}$*.*

Preuve

(a) Soit e une alarme finale dans A et $a \in \overline{e}$. Comme $a \sim e$, on a donc $a \rightsquigarrow e$ et $e \rightsquigarrow a$. Supposons que pour a, il y a une alarme $a' \in A$ qui dépend d'elle, c'est-à-dire $a \rightsquigarrow a'$. Par transitivité, $e \rightsquigarrow a'$. Et comme e est une alarme finale, $a' \rightsquigarrow e$, donc, $a' \rightsquigarrow a$ par transitivité aussi. Ce qui implique que a est aussi une alarme finale dans A.

(b) Soit $L \subseteq A$ et e une alarme finale dans A avec $e \in labels(p_L)$. Par la définition 4.1 de p_L, il existe un $l \in L$ tel que $e \rightsquigarrow l$. Comme e est une alarme finale, $l \rightsquigarrow e$. On conclut donc que $e \sim l$.

(c) découle immédiatement de (b) pour $L = \{a\}$.

(d) découle immédiatement de la définition 4.1 d'une *slice* appliquée à p_a et $p_{a'}$.

(e) découle immédiatement de la définition 4.1 d'une *slice* appliquée à p_a et $p_{a'}$. □

Nous aurons besoin des propositions définies dans le lemme 4.7.2 :

LEMME 4.7.2
Soit $A \subseteq alarms(p)$ un ensemble d'alarmes du programme p. Il existe une unique couverture minimale des alarmes par slicing de A. En d'autres termes, il existe un sous-ensemble $A' \subseteq A$ tel que :
(a) *$A = \bigcup_{a \in A'} labels(p_a) \cap A$, c'est-à-dire A' définit une couverture des alarmes de A par slicing,*
(b) *s'il existe un sous-ensemble $A'' \subseteq A$ qui définit une autre couverture des alarmes par slicing de A, alors $\text{card}(A'') \geq \text{card}(A')$ (c'est-à-dire A' est minimal).*
(c) *Si $A'' \subseteq A$ et A'' définit une autre couverture minimale des alarmes par slicing de A, les ensembles couvrants des deux couvertures sont identiques.*

PREUVE

(a) Nous montrons qu'il existe une *couverture des alarmes par slicing* de A. On choisit un représentant $e_i \in ends(A)$ dans chaque classe d'équivalence d'alarmes finales dans $ends(A/\sim)$. Soit $k = \text{card}(ends(A/\sim))$ et $A' = \{e_1, e_2, \ldots, e_k\}$ l'ensemble de ces représentants. Tout $a \in A$ a une alarme finale $e \in A$ qui dépend d'elle, dont la classe d'équivalence $\overline{e}$ a un représentant $e_j \in A'$. Comme $a \rightsquigarrow e$ et $e \rightsquigarrow e_j$, par transitivité, nous avons $a \rightsquigarrow e_j$, donc $a \in labels(p_{e_j}) \cap A$. On en déduit que $A = \bigcup_{a \in A'} labels(p_a) \cap A$. Donc A' définit une *couverture des alarmes par slicing* de A.

(b) Montrons maintenant la minimalité de la cardinalité de la *couverture des alarmes par slicing* A' de A. Supposons qu'il existe un sous-ensemble $A'' \subseteq A$ qui définit une autre *couverture des alarmes par slicing* de A, c'est-à-dire $A = \bigcup_{a \in A''} labels(p_a) \cap A$. Pour tout $j \in \{1, 2, \ldots, k\}$, on peut trouver $a_j \in A''$ tel que $e_j \in labels(p_{a_j}) \cap A$. Comme $e_j \in labels(p_{a_j})$ et e_j est une alarme finale dans A, on a $e_j \sim a_j$ d'après le lemme 4.7.1(c). En d'autres termes, $(a_1, \ldots, a_k)$ est une autre liste de représentants pour les différentes classes d'équivalence des alarmes finales $(\overline{e_1}, \ldots, \overline{e_k})$, ce qui implique que les éléments $a_1, a_2, \ldots, a_k$ sont tous différents. Nous avons trouvé au moins k éléments différents $a_1, a_2, \ldots, a_k$ dans A'', donc $\text{card}(A'') \geq k = \text{card}(A')$.

(c) Enfin, nous montrons l'unicité de la *couverture minimale des alarmes par slicing* de A. Supposons que $A'' \subseteq A$ et A'' définit une autre *couverture minimale des alarmes par slicing* de A. La preuve dans (a) ci-dessus a montré que A'' contient un sous-ensemble $\{a_1, a_2, \ldots, a_k\}$ où chaque a_j est un autre représentant de la classe d'alarmes finales $\overline{e_j}$ et les a_j sont tous différents. En appliquant la minimalité pour la *couverture minimale des alarmes par slicing* définie par A'', on obtient $\text{card}(A') \geq \text{card}(A'')$, d'où $\{a_1, a_2, \ldots, a_k\} = A''$ et $\text{card}(A') = \text{card}(A'')$. Comme $a_j \sim e_j$, d'après le lemme 4.7.1(d), $p_{e_j} = p_{a_j}$ et les ensembles couvrants $labels(p_{e_j}) \cap A$ et $labels(p_{a_j}) \cap A$ des deux couvertures sont identiques, pour tout $j \in \{1, 2, \ldots, k\}$. □

On a prouvé que toute *couverture minimale des alarmes par slicing* de A est définie par un ensemble de représentants des classes d'alarmes finales, et les ensembles couvrants

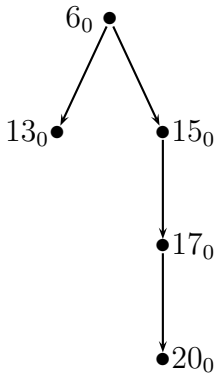


FIGURE 4.9 – Dépendances entre alarmes pour le programme `eurocheck`

sont définis de manière unique.

Pour l'exemple de la figure 4.4, les dépendances entre les alarmes sont illustrées dans la figure 4.9. Les alarmes 13_0 et 20_0 sont des alarmes finales et l'ensemble $\{13_0, 20_0\}$ définit une *couverture minimale des alarmes par slicing*. Les ensembles couvrants sont $\{6, 13\}$ et $\{6, 15, 17, 20\}$.

4.8 Slice & Test : options avancées

Dans cette partie, nous introduisons les options avancées de la fonction Slice & Test : *min* et *smart*, en nous basant sur les dépendances entre les menaces étudiées dans la partie 4.7. Soit A l'ensemble des alarmes de p.

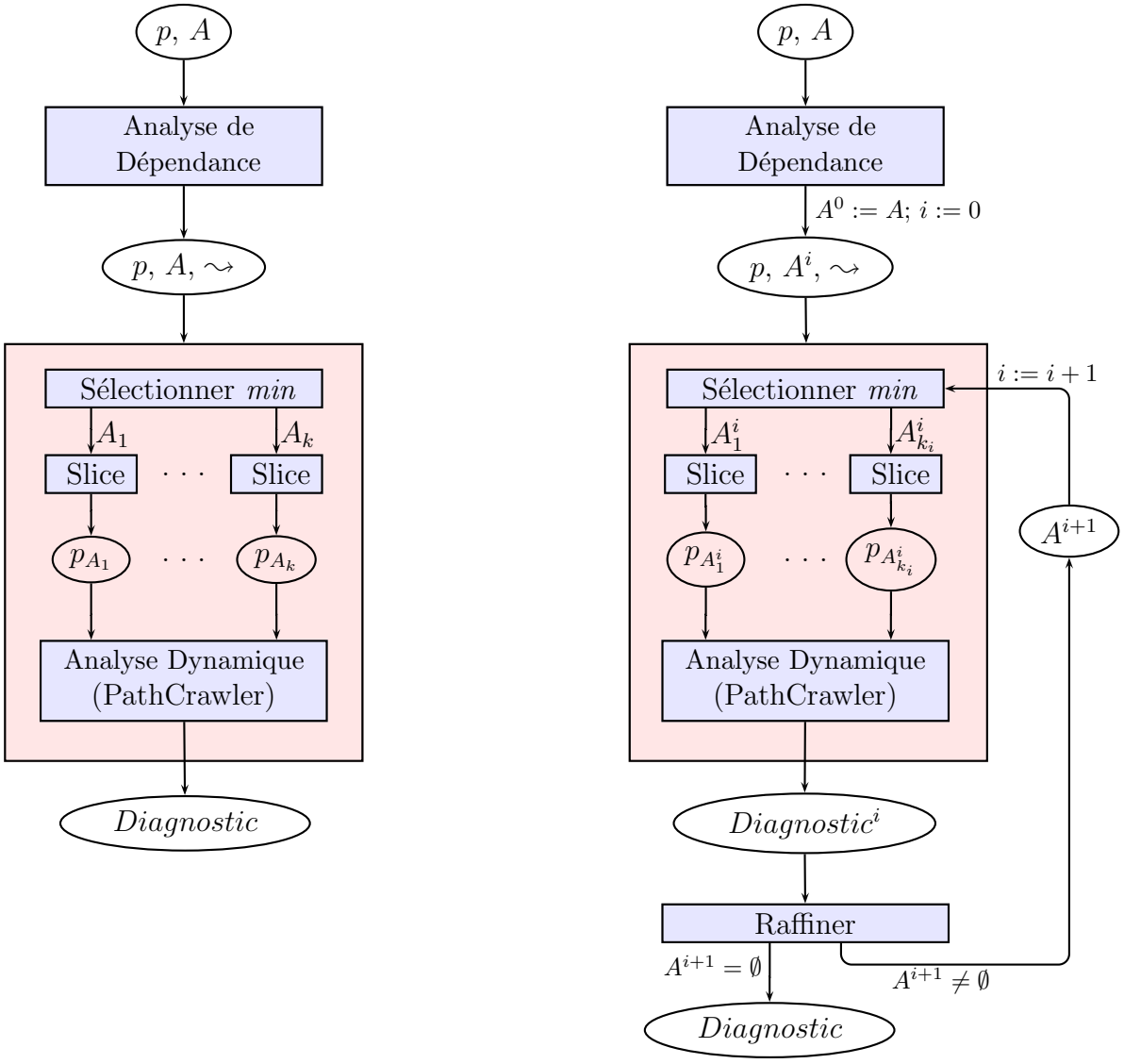
4.8.1 Option *min*

Cette option présente une solution intermédiaire entre les deux options *all* et *each* présentées précédemment (partie 4.6). Avec cette option (voir figure 4.10a), l'opération de *slicing* est effectuée k fois pour les ensembles couvrants $A_1, A_2, \ldots, A_k$ d'une *couverture minimale des alarmes par slicing* de A et les *slices* $p_{A_1}, p_{A_2}, \ldots, p_{A_k}$ sont produites telles que

$$\forall i \in \{1, 2, \ldots, k\},\ A_i \neq \emptyset\ et A_1 \cup A_2 \cup \ldots \cup A_k = A.$$

Ensuite, l'analyse dynamique est appliquée séparément à $p_{A_1}, p_{A_2}, \ldots, p_{A_k}$.

Les critères du *slicing* $A_1, A_2, \ldots, A_k$ sont choisis en fonction de dépendances entre les alarmes. Si toutes les alarmes sont dépendantes les unes des autres alors l'option *min* devient identique à l'option *all*. Si toutes les alarmes sont indépendantes, alors l'option *min* devient identique à l'option *each*.



a) Option *min* **b)** Option *smart*

FIGURE 4.10 – Slice & Test : options avancées

Nous avons montré dans le lemme 4.7.2 que ce type de couverture existe et que ses ensembles couvrants sont définis d'une manière unique. Techniquement, il suffit de choisir un ensemble de représentants $a_1, \ldots, a_k$ des alarmes finales de A, ensuite déterminer les ensembles A_i avec $A_i = \mathit{labels}(p_{a_i}) \cap A$ et calculer les *slices* p_{a_i} parce que $p_{a_i} = p_{A_i}$ d'après le lemme 4.7.1(e).

La fonction *min* décrite dans l'algorithme 1 procède ainsi. Elle initialise les ensembles S, B et U d'alarmes contenant les alarmes avec le statut respectivement, safe, bug et

Algorithme 1 : *min*

Entrées : p : programme
A : ensemble des alarmes de p
Sorties : S, B, U : ensembles d'alarmes respectivement safe, bug et unknown.
Données : $A_1, \ldots, A_k$ les ensembles couvrant de la couverture des alarmes de A par *slicing*
$\rightsquigarrow$ relation de dépendances sur A

```
1  S ← ∅;
2  B ← ∅;
3  U ← A;
4  ⇝← AnalyseDependance(p, A);
5  (A_1, ..., A_k) ← SelectionnerMin(A, ⇝);
6  for i ← 1 to k do
7      p_{A_i} ← Slice(p, A_i) ;
8      (S, B, U) ← (S, B, U) ⊎ AnalyseDynamique(p_{A_i}, A_i);
9  end
10 return (S, B, U);
```

unknown : S et B sont vides et U contient toutes les alarmes (lignes 1–3). Puis la fonction *AnalyseDependance* calcule les dépendances de la relation $\rightsquigarrow$ (ligne 4). L'algorithme calcule les ensembles couvrants $A_1, \ldots, A_k$ de la *couverture minimale des alarmes par slicing* de l'ensemble A, en appliquant la fonction *SelectionnerMin* (ligne 5). Puis, pour chaque ensemble A_i, $i \in \{1, \ldots, k\}$, le programme est simplifié par la fonction *Slice* et analysé dynamiquement par la fonction *AnalyseDynamique* (lignes 6–9). Cette dernière construit le diagnostic pour les alarmes de p à partir d'une analyse dynamique sur la *slice* p_{A_i} (voir la définition 2). Le diagnostic est cumulé par fusion de diagnostics à chaque étape. Le diagnostic final est retourné par la fonction (ligne 10).

Cette option combine les avantages des options de base *all* et *each* décrites dans la partie 4.6. En effet elle produit des *slices* plus simples que celle de l'option *all*, par conséquent, les alarmes les plus faciles à classer sont moins pénalisées par l'analyse des alarmes complexes. Le nombre de *slices* est au plus k, souvent beaucoup moins qu'avec *each*. De plus, chaque *slice* p_{A_i} est importante car elle peut permettre de classer le nœud feuille a_i qu'aucune autre *slice* p_{A_j} ne permettra de classer, donc l'analyse dynamique de p_{A_i} n'est jamais superflue.

Pour l'exemple de la figure 4.3, la figure 4.9 montre les dépendances entre les alarmes et la figure 4.11 montre les critères de *slicing* pour chacune des 4 méthodes : *all*, *each*, *min* et *smart*. Pour l'option *min*, les ensembles couvrants de la *couverture minimale des alarmes par slicing* sont $A_0 = \{6_0, 13_0\}$ et $A_1 = \{6_0, 15_0, 17_0, 20_0\}$. SANTE produit deux *slices* : p_{A_0} et p_{A_1}. Le temps complet requis par SANTE pour analyser les deux *slices* et classer les cinq alarmes est d'environ 6 secondes. Rappelons que ce temps était de 25 secondes pour *none*, de 7 secondes pour *all* et de 13 secondes pour *each*.

critères de *slicing* avec chaque option
all : $\{6_0, 13_0, 15_0, 17_0, 20_0\}$
each : $\{6_0\}, \{13_0\}, \{15_0\}, \{17_0\}, \{20_0\}$
min : $A_0 = \{6_0, 13_0\}$, $A_1 = \{6_0, 15_0, 17_0, 20_0\}$
smart : $A_1^0 = \{6_0, 13_0\}$, $A_2^0 = \{6_0, 15_0, 17_0, 20_0\}$ et si $6_0, 15_0, 17_0$ restent unknown, $A_1^1 = \{6_0, 15_0, 17_0\}$ et si $6_0, 15_0$ restent unknown, $A_1^2 = \{6_0, 15_0\}$ et si 6_0 reste unknown, $A_1^3 = \{6_0\}$

FIGURE 4.11 – Déroulement de l'étape Slice & Test pour le programme `eurocheck` avec les différentes options

La faiblesse de cette option apparaît lorsque l'analyse dynamique de p_{A_i} dépasse le temps limite sans classer certaines alarmes $a' \in A_i$, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple (par exemple $p_{a'}$) permettrait de classer a'. L'option suivante pallie cet inconvénient.

4.8.2 Option *smart*

Cette option (voir figure 4.10b) applique l'option *min* itérativement sur une séquence d'ensembles d'alarmes A^i dont la taille $\text{card}(A^i)$ diminue après chaque itération. Initialement, $i = 0$ et $A^0 = A$. Pour chaque $i \geq 0$, l'algorithme calcule la *couverture minimale des alarmes par slicing* $\{A_1^i, A_2^i, \ldots, A_{k_i}^i\}$ de l'ensemble A^i en appliquant la fonction *SelectionnerMin* et ensuite génère les *slices* correspondantes. L'analyse dynamique génère $Diagnostic^i$ pour A^i. Ensuite, la fonction *Raffiner* calcule A^{i+1} comme l'ensemble des alarmes dans $A^i \setminus ends(A^i)$ qui demeurent non classées (unknown) par $Diagnostic^i$. Notons que les alarmes finales de A^i sont explicitement exclues dans A^{i+1}, sinon nous aurions pu avoir $A^{i+1} = A^i$ et donc répéter la même opération. Enfin, on incrémente i et on répète l'itération pour le nouvel ensemble A^i jusqu'à ce que A^{i+1} devienne vide.

La fonction *smart* décrite dans l'algorithme 2 procède ainsi. Elle initialise les ensembles S, B et U d'alarmes avec le statut, repectivement, safe, bug et unknown. S et B sont vides et U contient toutes les alarmes (lignes 1–3). Puis la fonction *AnalyseDependance* calcule les dépendances $\rightsquigarrow$ (ligne 4) et l'ensemble d'alarmes à traiter dans la prochaine itération, appelé A^i, par toutes les alarmes (ligne 6). Tant que l'ensemble d'alarmes à traiter n'est pas vide, l'algorithme calcule les ensembles couvrants de la couverture minimale des alarmes de l'ensemble A^i par *slicing* $(A_1^i, \ldots, A_{k_i}^i)$ formés de k_i ensembles, en appliquant la fonction *SelectionnerMin* (ligne 8). Puis, pour chaque ensemble A_j^i, $j \in \{1, \ldots, k\}$, le programme est simplifié par la fonction *Slice* et analysé dynamiquement par la fonction *AnalyseDynamique* (lignes 9–12). Cette dernière (ligne 11) construit le diagnostic pour les alarmes de p à partir d'une analyse dynamique sur la *slice* $p_{A_j^i}$ (voir la définition 2).

Algorithme 2 : *smart*

Entrées : p : programme
A : ensemble des alarmes de p
Sorties : S, B, U : ensembles d'alarmes respectivement safe, bug et unknown.
Données : A^i : ensemble des alarmes traitées à chaque itération i
$A^i_1, A^i_2, \ldots, A^i_{k_i}$ les ensembles couvrants de la couverture minimale des alarmes par *slicing* de A^i
$\rightsquigarrow$ relation de dépendances sur A

```
1  S ← ∅;
2  B ← ∅;
3  U ← A;
4  ⇝← AnalyseDependance(p, A);
5  i ← 0;
6  A^i ← A;
7  while A^i ≠ ∅ do
8      (A^i_1, A^i_2, ..., A^i_{k_i}) ← SelectionnerMin(A^i, ⇝);
9      for j ← 1 to k do
10         p_{A^i_j} ← Slice(p, A^i_j) ;
11         (S, B, U) ← (S, B, U) ⊎ AnalyseDynamique(p_{A^i_j}, A^i_j);
12     end
13     A^{i+1} ← A^i − (ends(A^i) ∪ S ∪ B);
14     i ← i + 1;
15 end
16 return (S, B, U);
```

Le diagnostic est cumulé par fusion de diagnostics à chaque étape.

Ensuite, on calcule la valeur suivante de A^i (ligne 13) ce qui correspond à la fonction "Raffiner" dans la figure 4.10b. Le diagnostic final est retourné par la fonction (ligne 16).

Par exemple, dans la figure 4.11, $A^0 = A = \{6_0, 13_0, 15_0, 17_0, 20_0\}$ et l'ensemble $\{13_0, 20_0\}$ définit une *couverture minimale des alarmes par slicing* de A^0. Les ensembles couvrants sont $A^0_1 = \{6_0, 13_0\}$ et $A^0_2 = \{6_0, 15_0, 17_0, 20_0\}$. Si l'analyse dynamique de $p_{A^0_1}$ et $p_{A^0_2}$ ne classe pas 6_0, 15_0 et 17_0, les deux alarmes finales 13_0 et 20_0 sont retirées de A^0 pour la prochaine itération ($i = 1$) et $A^1 = \{6_0, 15_0, 17_0\}$. Pour A^1, l'ensemble $\{17_0\}$ définit une *couverture minimale des alarmes par slicing* et l'ensemble couvrant est $A^1_1 = \{6_0, 15_0, 17_0\}$ De nouveau, si l'analyse dynamique de $p_{A^1_1}$ ne classe pas 6_0 et 15_0, seule l'alarme finale 17_0 est retirée de A^1, et $A^2 = A^2_1 = \{6_0, 15_0\}$ pour la prochaine itération ($i = 2$). De nouveau si l'analyse dynamique de $p_{A^2_1}$ ne classe pas 6_0, seule l'alarme finale 15_0 est retirée de A^2, et $A^3 = A^3_1 = \{6_0\}$ pour la prochaine itération ($i = 3$) qui sera la dernière.

Lorsque A^{i+1} devient vide, le diagnostic final classe $a \in A$ comme safe (respectivement

bug) si au moins un $Diagnostic^i$ classe a comme safe (respectivement bug), sinon a reste non classée (unknown).

Avec cette option, chaque alarme est analysée séparément (autant que possible) par analyse dynamique, et c'est fait *juste quand c'est nécessaire*, c'est-à-dire lorsqu'elle ne peut être classée par l'analyse dynamique d'une plus grande *slice*. Cette option permet donc d'éviter la redondance de *each* et de résoudre le problème du manque de temps potentiel qui se pose avec *min*.

Pour l'exemple de la figure 4.3, la première itération applique l'option *min* sur l'ensemble de toutes les alarmes $A^0 = \{6_0, 13_0, 15_0, 17_0, 20_0\}$ et réussit à classer les cinq alarmes en 6 secondes environ. D'autres itérations ne sont pas nécessaires. Si on suppose que cette première itération n'a classé aucune alarme, la fonction *Raffiner* calcule alors $A^1 = A^0 \setminus ends(A^0) = \{6_0, 15_0, 17_0\}$ ($ends(A^0) = \{13_0, 20_0\}$ voir figure 4.9) et la deuxième itération applique l'option *min* sur A^1 et ainsi de suite jusqu'à ce que A^i devienne vide.

4.9 Correction de la méthode

Dans cette partie, nous montrons que la méthode SANTE est correcte, mais qu'elle est en général incomplète. Nous commençons par étudier la relation entre une alarme dans le programme initial et son diagnostic obtenu par DA sur une version simplifiée du programme (une *slice*).

Si la fonction d'analyse dynamique DA appliquée sur une *slice* retourne le diagnostic (bug, s) pour une alarme $a \in A$, alors, selon la définition 2, le diagnostic pour l'alarme a dans p peut devenir (bug, s) seulement après une confirmation par une exécution de p sur s. La nécessité de cette vérification a été expliquée dans la partie 4.5.3. Montrons maintenant que pour les alarmes classées safe sur une *slice*, aucune vérification supplémentaire n'est nécessaire.

LEMME 4.9.1
Soit A l'ensemble des alarmes d'un programme p, $A' \subseteq A$, $a \in A'$, et $p_{A'}$ la slice de p par rapport à A'. Si la fonction d'analyse dynamique DA appliquée sur $p_{A'}$ retourne le diagnostic safe pour l'alarme a, alors il n'y a aucun risque dans p en ce qui concerne l'alarme a.

PREUVE
Soit $a = (l, c)$ une alarme qui signale que l'instruction l est menaçante avec la condition d'erreur c.

Supposons que DA appliquée sur $p_{A'}$ retourne le diagnostic safe pour l'alarme $a \in A'$. Cela signifie qu'il n'y a aucune trajectoire dans $p_{A'}$ qui conduit à une erreur qui confirme

l'alarme a. La preuve sera faite par l'absurde : supposons que p n'est pas safe par rapport à l'alarme a, c'est-à-dire qu'il existe dans p, une trajectoire T

$$T = (l_1, s_1)(l_2, s_2)\dots(l_x, s_x)\dots ,$$

qui commence sur un état d'entrée $s_1 = s$ et qui provoque l'erreur signalée par l'alarme $a = (l, c)$ à l'instruction $l_x = l$ exécutée sur l'état s_x. Cela signifie que la condition d'erreur c est satisfaite sur $\hat{s} = s_x{\downarrow}\mathrm{ref}(l)$.

Soit

$$T' = (l'_1, s'_1)(l'_2, s'_2)\dots$$

la trajectoire dans $p_{A'}$ sur le même état d'entrée $s'_1 = s$. Considérons la trajectoire partielle

$$U = (l_1, s_1)(l_2, s_2)\dots(l_x, s_x)$$

de T. D'après la proposition 4.4.1(b), il existe une trajectoire partielle

$$U' = (l'_1, s'_1)(l'_2, s'_2)\dots(l'_q, s'_q)$$

de T' telle que $\mathrm{Proj}_{A'}(U) = \mathrm{Proj}_{A'}(U')$ et donc il existe un élément (l'_y, s'_y) dans la trajectoire U' $(1 \leq y \leq q)$ tel que (l_x, s_x) et (l'_y, s'_y) sont projetés sur la même paire non-vide :

$$\mathrm{Proj}_{A'}(l_x, s_x) = (l, \hat{s}) = \mathrm{Proj}_{A'}(l'_y, s'_y).$$

Par conséquent, $s_x \downarrow \mathrm{ref}(l) = \hat{s} = s'_y \downarrow \mathrm{ref}(l)$. Et comme l'erreur se produit à l'élément (l_x, s_x) de la trajectoire U, la condition d'erreur c est satisfaite sur $\hat{s}$, donc une erreur du même type se produit également à l'élément (l'_y, s'_y) de la trajectoire U' avec le même état d'entrée s, ce qui contredit l'hypothèse. Cela implique qu'il n'y a pas de risque dans p concernant l'alarme a. □

Deux résultats sont à déduire sur la correction de la méthode SANTE pour des programmes dont l'analyse se termine dans un temps fini. Ces résultats sont énoncés dans le théorème suivant.

THEORÈME 4.9.1 (CORRECTION DE SLICE & TEST)
(i) *la classification des alarmes dans l'étape Slice & Test de la méthode* SANTE *est correcte.*
(ii) *la classification des alarmes dans l'étape Slice & Test de la méthode* SANTE *est en général incomplète. Elle devient complète lorsque chaque alarme est classée comme* bug *ou* safe *sur au moins une slice qui la contient.*

PREUVE
Soit $A = \mathit{alarms}(p)$ l'ensemble des alarmes signalées par l'analyse de valeurs appliquée à un programme p. Soit $m \geq 1$ un entier et $\{A_1, A_2, \dots, A_m\} = A$ la couverture utilisée à l'étape Slice & Test de la méthode SANTE. Le diagnostic final de Slice & Test est obtenu par fusion des diagnostics pour les alarmes de p construits à partir des résultats d'analyse

dynamique sur les progammes simplifiés $p_{A_1}, p_{A_2}, \ldots, p_{A_m}$ (définition 2).

(i) La correction est garantie par les deux propriétés suivantes déduites directement des définitions 3 et 2 et du lemme 4.9.1 :

(a) Pour une alarme $a \in A$, s'il existe $i \in \{1, 2, \ldots, m\}$ tel que le diagnostic construit à partir des résultats d'analyse dynamique sur le progamme simplifié p_{A_i} retourne (bug, s) pour l'alarme a dans p, alors l'exécution de p sur l'état d'entrée s mène réellement à une erreur (voir la définition 2).

(b) Pour une alarme $a \in A$, s'il existe $i \in \{1, 2, \ldots, m\}$ tel que le diagnostic construit à partir des résultats d'analyse dynamique sur le progamme simplifié p_{A_i} retourne safe pour l'alarme a dans p, alors il n'y a pas de risque d'erreur dans p concernant l'alarme a (voir lemme 4.9.1).

(ii) La classification des alarmes dans l'étape Slice & Test est incomplète lorsque tous les diagnostics pour les alarmes de p construits à partir des résultats d'analyse dynamique sur les progammes simplifiés laissent au moins une alarme non classée. Cela peut être le cas pour une alarme $a \in A$ si pour toute *slice* p_{A_i} qui la contient, DA sur la *slice* p_{A_i} laisse l'alarme a non classée ou la classe comme un bug qui ne se confirme pas à l'exécution de p. La complétude se déduit également des définitions 3 et 2 et du lemme 4.9.1.

(c) Si, pour toute alarme $a \in A$, il existe au moins un $i \in \{1, 2, \ldots, m\}$ tel que le diagnostic pour les alarmes de p construit à partir des résultats d'analyse dynamique sur le programme simplifié p_{A_i} retourne le diagnostic safe ou bug pour l'alarme a, alors l'alarme a dans p est classée comme safe ou bug. □

Nous sommes prêts à énoncer le résultat sur la correction de la méthode SANTE.

THEORÈME 4.9.2 (CORRECTION DE SANTE)
(i) *La classification des menaces par la méthode* SANTE *est correcte.*
(ii) *La classification des menaces par la méthode* SANTE *est en général incomplète. Elle devient complète lorsque chaque alarme est classée par le diagnostic pour les alarmes de p construit à partir des résultats d'analyse dynamique sur au moins une slice.*

PREUVE
D'après la proposition 4.3.1, si une menace potentielle n'est pas signalée comme une alarme par l'analyse de valeurs, alors il n'y a pas d'erreur possible due à cette menace. Par définition, la méthode de SANTE ne considère pas de telles menaces et considère uniquement les alarmes signalées.

Les alarmes signalées par l'analyse de valeurs peuvent être classées comme bug ou safe par l'étape de Slice & Test. D'après le théorème 4.9.1, cette classification est correcte : une menace classée comme bug est vraiment un bug, et une menace classée comme safe ne

présente pas un risque d'erreur. Toutefois, certaines alarmes peuvent rester non classées (unknown). Ce qui induit l'incomplétude de la méthode. □

4.10 Synthèse

Dans ce chapitre, nous avons présenté de manière rigoureuse et détaillée la méthode SANTE qui combine l'analyse de valeurs, l'analyse de dépendances, le *slicing* et la génération de test structurel. Nous avons expliqué chaque étape de la méthode et nous les avons illustrées sur le programme `eurocheck`. Nous avons introduit les différentes options qui correspondent aux différentes utilisations du *slicing* :

- ***none*** : Le programme est directement analysé par analyse dynamique sans aucune simplification.

- ***all*** : Le *slicing* est appliqué une fois par rapport à l'ensemble de toutes les alarmes de p. Nous obtenons une version simplifiée contenant les mêmes menaces que le programme original p. Cependant, l'analyse dynamique peut manquer de temps ou d'espace et les alarmes qui sont plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes.

- ***each*** : Le *slicing* est effectué une fois par rapport à chaque alarme, produisant un nombre de programmes simplifiés égal au nombre d'alarmes présentes dans p. Pour chaque alarme, on produit le programme simplifié minimal. Cependant, l'analyse dynamique est exécutée pour chaque programme et il y a un risque de redondance si des alarmes sont incluses dans plusieurs *slices* à cause des dépendances.

- ***min*** : L'opération de *slicing* est effectuée pour k ($k \leq n$ où n est le nombre d'alarmes) sous-ensembles d'alarmes couvrant toutes les alarmes présentes dans p. On a moins de *slices* qu'avec l'option *each*, et des *slices* plus simples qu'avec l'option *all*. Cependant, l'analyse dynamique de certaines *slices* peut manquer de temps ou d'espace avant de classer certaines alarmes, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple permettrait de les classer.

- ***smart*** : Cette option applique l'option *min* itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire. Lorsqu'une alarme ne peut pas être classée par l'analyse dynamique d'une *slice*, des *slices* plus simples sont calculées.

Ce travail a été publié dans l'article [CKGJ10b] où nous avons présenté la première version de SANTE qui combine l'analyse statique et la génération de tests structurels uniquement. Dans l'article [CKGJ11] nous avons introduit l'utilisation du *slicing* dans SANTE et les deux options de base (*all* et *each*). L'article [CKGJ12] présente les options avancées (*min* et *smart*). Le chapitre suivant s'intéresse à l'implémentation de la méthode.

Chapitre 5

Implémentation

Sommaire

5.1 Introduction

Dans le chapitre 4, nous avons présenté la méthode SANTE, qui combine l'analyse de valeurs, le *slicing* et l'analyse dynamique. Pour évaluer l'apport de cette méthode, nous avons implémenté un prototype nommé SANTE CONTROLLER. Il se base sur les deux outils FRAMA-C et PATHCRAWLER présentés dans le chapitre 3.

FRAMA-C, présenté dans la partie 3.4, regroupe plusieurs analyseurs statiques dans un cadre de collaboration qui permet à des analyseurs statiques de s'appuyer sur les résultats déjà calculés par d'autres analyseurs. Parmi ses analyseurs, FRAMA-C fournit une

analyse de valeurs, une analyse de dépendances et un outil de *slicing*.

PathCrawler, présenté dans la partie 3.3, est un outil de génération de tests structurels pour les fonctions C. Il analyse le code source pour déduire les cas de test. Il cherche à satisfaire le critère de couverture de "tous les chemins", ou le critère de couverture de "tous les k-chemins" (k-*path*), qui restreint la génération aux chemins avec au plus k itérations successives de chaque boucle. Il utilise la recherche en profondeur d'abord.

Pour implémenter notre méthode combinant l'analyse de valeurs, le *slicing* et l'analyse dynamique, nous utilisons les greffons d'analyse de valeurs, d'analyse de dépendances et de *slicing* de Frama-C et PathCrawler pour l'analyse dynamique.

Le chapitre s'organise comme suit : La partie 5.2 présente le travail effectué pour rapprocher les deux outils, les difficultés d'intégration et les solutions adoptées. La partie 5.3 présente la traduction des conditions d'erreurs pour chaque type de menaces traitées. Des adaptations techniques visant à traiter les cas d'erreurs dans PathCrawler sont énoncées dans la partie 5.4. La partie 5.5 explique comment sont calculés les ensembles couvrants d'une couverture des alarmes par *slicing*, qui sont définis en partie 4.7. Le greffon sante controller qui contrôle les différentes analyses et fournit les diagnostics finaux est présenté dans la partie 5.6. Une synthèse conclut ce chapitre.

5.2 Rapprochement de Frama-C et PathCrawler

Les outils Frama-C et PathCrawler sont développés indépendamment l'un de l'autre et n'ont pas été initialement conçus pour collaborer. sante assemble ces deux outils hétérogènes, utilisant différentes technologies telles que l'interprétation abstraite et la programmation en logique avec contraintes. Dans cette partie, nous décrivons d'une part l'intégration de PathCrawler dans Frama-C par le greffon *PC Analyzer* et d'autre part l'adaptation de la génération de test dans PathCrawler afin de détecter des erreurs à l'exécution à partir des assertions ACSL.

Frama-C est extensible par construction, grâce à son architecture à greffons. Chaque greffon peut utiliser les résultats d'analyses faites par les autres greffons. ainsi que les fonctionnalités fournies par le noyau même de Frama-C. Notre choix d'implémentation est donc de relier PathCrawler à Frama-C via un nouveau greffon dans Frama-C, et d'adapter PathCrawler afin qu'il accepte les informations fournies par d'autres greffons.

Comme PathCrawler, le noyau de Frama-C se base sur le format normalisé CIL [CIL06]. Mais Frama-C étend CIL avec le langage de spécification ACSL (ANSI / ISO C Specification Language) [BFH+08]. Cette extension de CIL étend l'interface d'utilisation ainsi que les types et les opérations afin de traiter correctement les programmes C annotés. Par conséquent, la version de CIL utilisée dans Frama-C et celle qui était utilisée dans PathCrawler avant cette thèse (avant la version 2.8) n'étaient pas compatibles

```
1 let startup _ =
2   if Options.Enabled.get () then
3     begin
4       Format.printf "@\n[PCAnalyzer] started...@.";
5       let proj = createProject () in
6       let ast = copyAstToProject proj in
7       try
8         visitFunctions ast;
9         generatePifFiles;
10        generateLanceur;
11        generateInstrumentedCode;
12        Format.printf "@\n[PCAnalyzer] finished...@.";
13      with Exit -> ()
14  end
```

FIGURE 5.1 – Fonction principale de *PC Analyzer*

et les arbres de syntaxe abstraite fournis par les deux étaient complètement différents.

5.2.1 PC Analyzer

Nous avons développé un greffon de FRAMA-C nommé *PC Analyzer* (PATHCRAWLER Analyzer). Il est écrit en OCAML et est désormais utilisé dans toutes les distributions de PATHCRAWLER, depuis la version 2.8 de février 2010. Ce greffon réalise l'étape d'analyse et d'instrumentation, la première étape dans le fonctionnement de PATHCRAWLER, qui consiste à instrumenter le programme sous test et à générer les fichiers nécessaires à l'étape de génération de tests. Ces derniers sont générés à partir de l'arbre de syntaxe abstraite fourni par FRAMA-C.

PC Analyzer exploite les services fournis par le noyau de FRAMA-C. Il est également bien placé pour exploiter les analyseurs distribués avec FRAMA-C. *PC Analyzer* adapte également PATHCRAWLER afin qu'il accepte les informations fournies par d'autres greffons.

La figure 5.1 présente sommairement la fonction principale de *PC Analyzer*. La conditionnelle à la ligne 2 vérifie si l'option *PC Analyzer* est sélectionnée. Les lignes 4 et 12 marquent le début et la fin du traitement. À la ligne 5, la fonction `createProject` utilise les primitives fournies par le noyau de FRAMA-C pour créer un nouveau projet d'analyse. La fonction `copyAstToProject` (ligne 6) utilise aussi les primitives fournies par le noyau de FRAMA-C. Elle copie l'arbre de syntaxe normalisé dans ce projet et retourne cet arbre. À la ligne 8, la fonction `visitFunctions` parcourt l'arbre de syntaxe (`ast`) pour collecter les informations nécessaires pour créer les fichiers PIF (voir la section 3.3.3). Cette fonction utilise le visiteur `pathcrawlerVisitor` que nous avons développé à cet effet et qui hérite du visiteur fourni par le noyau de FRAMA-C. La fonction `generatePifFiles`

(ligne 9) génère les fichiers PIF (regroupés sous le nom `f.pif` dans la figure 3.5) ainsi que le fichier de précondition par défaut (nommé `params` dans la figure 3.5). La fonction `generateLanceur` (ligne 10) génère le lanceur (nommé `lanceur.c` dans la figure 3.5). La fonction `generateInstrumentedCode` (ligne 11) génère le fichier instrumenté (nommé `inst.c` dans la figure 3.5).

La mise en œuvre du *PC Analyzer* ouvre la voie à d'autres pistes de recherche de combinaisons potentielles entre l'analyse dynamique faite par PathCrawler et les autres greffons d'analyse statique de Frama-C, notamment le WP et le RTE.

5.2.2 Traitement des assertions dans PathCrawler

PC Analyzer apporte une partie du langage de spécification ACSL à PathCrawler. Actuellement *PC Analyzer* implémente un sous ensemble d'ACSL, notamment les assertions avec le mot clé `assert`. Des travaux en cours au laboratoire de sûreté des logiciels (LSL) au CEA LIST visent à implémenter les préconditions ACSL dans *PC Analyzer*.

Le traitement d'assertions dans *PC Analyzer* correspond à l'ajout des branches d'erreurs présenté dans la partie 4.5.1. Ces branches étendent les capacités de l'outil pour lui permettre de rechercher les cas de test atteignant une situation d'erreur à l'exécution (accès à la mémoire invalide, division par zéro, débordement de buffer, ...). Pour cela nous intégrons dans le programme à tester des objectifs de test supplémentaires représentant ces situations.

Ce traitement des assertions dans PathCrawler étend son domaine d'utilisation. PathCrawler était initialement utilisé pour vérifier des propriétés structurelles, par exemple l'absence de code mort ou encore la couverture des instructions par un ensemble de tests et des propriétés de performance, par exemple le pire temps d'exécution (WCET) d'un programme C. L'ajout des assertions étend PathCrawler afin qu'il puisse vérifier des propriétés de sûreté telles que l'absence d'erreurs à l'exécution. Il peut également générer des cas de test exhibant des violations d'assertions insérées dans le programme par l'utilisateur.

5.3 Traduction des conditions d'erreurs

L'analyse de valeurs de Frama-C marque chaque instruction menaçante du programme sous test par une assertion placée juste avant. Cette assertion signale l'alarme. Elle est de la forme :

```
//@ assert(Cond);
```

où `Cond` exprime en ACSL la condition qui empêche l'erreur.

```
1 int Division(int x, int z){
2   int y;
3   y = x/z;
4   return y;
5 }
```

FIGURE 5.2 – L'exemple Division

Afin de forcer la génération de tests à activer cette menace, nous ajoutons des branches d'erreur dans le code source. L'ajout de ces branches d'erreurs consiste à remplacer l'instruction menaçante, notée ci-dessous

```
threatStatement;
```

par une conditionnelle avec la condition d'erreur. Si cette dernière est vérifiée, la fonction `error` est appelée pour arrêter l'exécution du cas de test en cours et signaler la violation de l'assertion. La conditionnelle est de la forme :

```
if(Cond′)
  error();
else
  threatStatement;
```

où `Cond′` est l'expression en C de la condition ¬`Cond` qui provoque l'erreur.

`Cond′` est la négation de `Cond`. En pratique, `Cond` est exprimée en ACSL et ne peut pas toujours être exécutée directement. C'est pourquoi nous devons traduire ¬`Cond` en une condition d'erreur exprimée en C et exécutable. On traite trois cas :

- une alarme de division par zéro,
- une alarme d'accès hors limite dans un tableau de taille fixe,
- une alarme de pointeur invalide ou d'accès hors limite dans un tableau de taille variable.

5.3.1 Division par zéro

Si, pour une division, l'analyse de valeur ne peut pas garantir que l'expression au dénominateur est différente de zéro, elle ajoute une assertion avec, comme condition, l'expression au dénominateur différente de zéro.

Pour l'exemple `Division` présenté dans la figure 5.2, si l'analyse de valeur ne peut pas garantir que `z` est différente de zéro, elle ajoute l'assertion :

```
//@assert(z!=0);
```

Dans ce cas, la traduction de la condition est directe. Il suffit de prendre la négation de la condition `Cond` donnée par l'assertion. Dans l'exemple, l'instruction conditionnelle qui remplace l'instruction menaçante est :

```
1 int T[10];
2 int elemInArray(int a){
3   int y;
4   y = T[a];
5   return y;
6 }
```

FIGURE 5.3 – Exemple avec un tableau de taille fixe

```
if(!(z!=0))
  error();
else
  y = x/z;
```

5.3.2 Accès hors limite dans un tableau (local ou global) de taille fixe

Pour un accès tableau en général, si l'analyse de valeur ne peut pas garantir que c'est un accès valide, c'est-à-dire que la valeur consultée ou modifiée entre dans les bornes du tableau, elle ajoute une assertion avec, comme condition, la condition qui empêche l'accès hors limite. Dans le cas d'un tableau de taille fixe, l'analyse de valeur connaît la taille du tableau et il est facile d'établir la condition de sûreté entre l'indice d'accès et la taille du tableau correspondant.

Pour l'exemple `elemInArray` présenté dans la figure 5.3, l'analyse de valeur ne peut pas garantir que `a` est un indice valide dans `T`. Il ajoute donc l'assertion :

```
//assert(0 <= a ∧ a < 10);
```

avant la ligne 4.

Dans ce cas aussi, la traduction de la condition est directe. Il suffit de prendre la négation de la condition donnée par l'assertion. L'instruction conditionnelle qui remplace l'instruction menaçante est :

```
if(!(0 <= a && a < 10))
  error();
else
  y = T[a];
```

```
1 char elemInStr(char *str, int a){
2   char y;
3   y = *(str+a);
4   return y;
5 }
```

FIGURE 5.4 – Exemple avec un accès pointeur

```
1  char elemInStr(char *str, int a){
2    char y;
3₁   if( pathcrawler_length(str) <= a || a < 0 )
3₂       error();
3₃   else
3      y = *(str+a);
4    return y;
5  }
```

FIGURE 5.5 – L'exemple de la figure 5.4 après l'ajout des branches d'erreur

5.3.3 Pointeur invalide ou accès hors limite dans un tableau de taille variable

On considère dans cette partie les tableaux de taille variable et les pointeurs globaux ou locaux ainsi que tout tableau ou pointeur en entrée d'une fonction. En effet, la taille d'un tableau en entrée est perdue par CIL, conformément à la norme C.

Pour un accès par pointeur, ou dans un tableau de taille variable, si l'analyse de valeur ne connaît pas la taille exacte du tableau consulté, et trouve un risque d'accès invalide, elle ajoute une assertion avec, comme condition, la condition qui empêche l'accès invalide. Mais, dans ce cas, elle utilise la commande ACSL `\valid`.

Pour l'exemple `elemInStr` présenté dans la figure 5.4, si l'analyse de valeur ne peut pas garantir que `a` est un indice valide dans `str`, elle ajoute avant la ligne 3 l'assertion :

```
//@assert \valid(str + a);
```

Dans ce cas, la condition qui provoque l'erreur dépend de la taille allouée pour `str` dans chaque cas de test. Cette taille fait partie des données de test qui doivent être déterminées par PATHCRAWLER. Nous allons voir comment nous étendons PATHCRAWLER afin de lui permettre de récupérer la taille des tableaux donnés en entrée, à l'aide de la fonction nommée `pathcrawler_length` décrite ci-dessous.

Grâce à cette extension, la condition d'erreur sera simplement remplacée par :

```
pathcrawler_length(str) <= a || a < 0
```

```
1 struct __pathcrawler_array {
2   void* ptr;
3   size_t length;
4   size_t element_size;
5   struct __pathcrawler_array* next;
6 };
7 struct __pathcrawler_array* __pathcrawler_first_array = NULL;
```

FIGURE 5.6 – La structure `__pathcrawler_array`

ce qui la rend exécutable lors de l'exécution concrète et fera le lien avec la taille du tableau alloué lors de l'exécution symbolique. Le programme après l'ajout des branches d'erreur est présenté dans la figure 5.5.

Extension pour la taille des tableaux passés en paramètres

Pour traiter les alarmes sur les tableaux (et pointeurs) en entrée de la fonction sous test, on a besoin de connaître la taille du tableau donné en entrée. Cependant, le langage C ne nous permet pas de connaître la taille des tableaux donnés en entrée d'une fonction. En effet, la taille d'un tableau alloué dynamiquement n'est pas récupérable dans le code C après son allocation.

Rappelons que la fonction `sizeof` donne bien le nombre d'octets occupés par un tableau de taille fixe. Mais un tableau de taille variable est considéré comme un pointeur, dont la taille est de quatre octets utilisés pour stocker l'adresse de la zone pointée (sur une machine 32 bits).

Nous étendons PATHCRAWLER afin de lui permettre de récupérer la taille des tableaux en entrée d'une fonction, en le dotant de la fonction `pathcrawler_length`. En effet, nous surchargeons les fonctions d'allocation et de libération de mémoire afin de sauvegarder la taille allouée à chaque fois. La fonction `pathcrawler_length` prend un pointeur en entrée et permet de récupérer la taille allouée pour ce pointeur passé en paramètre. Dans ce qui suit, nous expliquons les modifications apportées en détails.

Parmi les rôles du lanceur, il y a l'allocation de la mémoire nécessaire aux entrées de la fonction sous test (voir section 3.3.3). Nous lui ajoutons une nouvelle fonctionnalité, qui consiste à enregistrer la taille des tableaux alloués dynamiquement par le lanceur avant un cas de test, afin que nous puissions la restituer ensuite par la fonction `pathcrawler_length`. Pour cela, nous modifions l'implémentation des deux fonctions `pathcrawler_alloc` et `pathcrawler_free`, dont les rôles initiaux étaient respectivement d'allouer et de libérer la zone mémoire nécessaire pour chaque paramètre.

Nous ajoutons une variable globale `__pathcrawler_first_array` qui pointe vers une

```
1 void* pathcrawler_alloc(int length, int element_size) {
2   struct __pathcrawler_array* elem;
3   elem = malloc(sizeof(struct __pathcrawler_array));
4   if (elem == 0)
5     return 0;
6   void* ptr = malloc(length * element_size);
7   if (ptr == 0){
8     free(elem);
9     return 0;
10  }
11  struct __pathcrawler_array** pnext = & __pathcrawler_first_array;
12  while (*pnext && (*pnext)->ptr < ptr)
13    pnext = & ((*pnext)->next);
14  elem->ptr = ptr;
15  elem->next = *pnext;
16  elem->length = length;
17  elem->element_size = element_size;
18  *pnext = elem;
19  return ptr;
20 }
```

FIGURE 5.7 – La fonction `pathcrawler_alloc`

liste chaînée contenant les descripteurs des tableaux alloués. Chaque élément de la liste instancie la structure `__pathcrawler_array` présentée dans la figure 5.6. Cette structure contient le descripteur d'un tableau alloué et inclut quatre champs :

- le premier contient l'adresse de la zone mémoire allouée pour le tableau,
- le champ `length` correspond au nombre d'éléments dans ce tableau,
- le champ `element_size` correspond à la taille de chaque élément et
- le champ `next` fait référence au prochain élément de la liste chaînée.

Notre nouvelle implémentation de `pathcrawler_alloc` est présentée dans la figure 5.7. Elle commence par allouer un élément pour stocker un nouveau descripteur de tableau (ligne 3). Ensuite on alloue la mémoire nécessaire pour le tableau (ligne 6). On cherche la place du descripteur dans la liste (lignes 12–13). On initialise les informations du descripteur (lignes 14–17). On place le descripteur dans la liste et on retourne le tableau alloué.

Notre nouvelle implémentation de `pathcrawler_free` est présentée dans la figure 5.8. Elle cherche le descripteur correspondant à ce tableau dans la liste des descripteurs (lignes 3–4). La ligne 7 relie l'élément précédent à l'élément suivant de descripteur du tableau à libérer. On libère le descripteur (ligne 8) et ensuite on libère la zone mémoire allouée pour le tableau (lignes 9–10).

La fonction `pathcrawler_length` est présentée dans la figure 5.9. Elle cherche le des-

```
1 void* pathcrawler_free(void* ptr)) {
2   struct __pathcrawler_array** pnext = &__pathcrawler_first_array;
3   while (*pnext != NULL && (*pnext)->ptr < ptr)
4     pnext = &((*pnext)->next);
5   if (*pnext != NULL && (*pnext)->ptr == ptr) {
6     struct __pathcrawler_array* current = *pnext;
7     *pnext = current->next;
8     free(current);
9     if (ptr != NULL)
10      free(ptr);
11  }
12  return;
13 }
```

FIGURE 5.8 – La fonction `pathcrawler_free`

```
1 int pathcrawler_length(void* ptr) {
2   struct __pathcrawler_array* cur = __pathcrawler_first_array;
3   while (cur != NULL && cur->ptr < ptr) {
4     cur = cur->next;
5   }
6   int len = 0;
7   if (cur != NULL && cur->ptr == ptr) {
8     len = cur->length;
9   }
10  return len;
11 }
```

FIGURE 5.9 – La fonction `pathcrawler_length`

cripteur correspondant au tableau en entrée, dans la liste des descripteurs des tableaux alloués (lignes 2–7). Une fois trouvé, on retourne la taille du tableau (ligne 10).

L'appel de la fonction `pathcrawler_length` ne doit pas être traité par le moteur d'exécution symbolique comme un appel ordinaire à une autre fonction. Nous avons donc modifié l'étape d'analyse et d'instrumentation pour traduire l'appel à `pathcrawler_length` par un prédicat prolog `dim`.

Par exemple, pour la première sous-condition de la ligne 3_1 de la figure 5.5, la nouvelle traduction PIF est :

```
block(+ID, cond(supegal, 'a', dim(str)), pos, (3_1, 1, 'euchk.c')).
block(-ID, cond(inf, 'a', dim(str)), neg, (3_1, 1, 'euchk.c')).
```

Ici `pathcrawler_length(str)` est directement traduit en PIF par le prédicat `dim(str)` représentant symboliquement la taille de `str`, sans modéliser l'appel de la fonction `pathcrawler_length`.

Le prédicat `dim` existait déjà dans PIF, mais son utilisation était restreinte à la précondition. Donc il nous a fallu également modifier le moteur de l'exécution symbolique pour interpréter ce prédicat en contraintes au moment de l'évaluation symbolique du préfixe de chemin.

5.4 Traitement des exceptions

Afin de capter et signaler un bogue, nous avons ajouté un mécanisme d'exceptions dans PathCrawler. Le besoin était d'arrêter l'exécution du cas de test avant d'atteindre l'instruction menaçante avec un état d'erreur d'une part et d'autre part de retourner au lanceur pour exécuter les cas de test suivants.

Cela ne peut pas être fait avec de simples instructions comme `exit` ou `return`. La fonction `exit` arrête bien l'exécution du programme sous test, mais elle arrête aussi le lanceur qui a appelé la fonction sous test, alors qu'on aimerait continuer à utiliser ce lanceur pour recevoir et exécuter les cas de test suivants. Un `return` permet de retourner le contrôle dans la fonction appelante qui n'est pas forcément le lanceur parce que la fonction sous test peut appeler d'autres fonctions.

Donc nous avons implémenté un mécanisme d'exception ressemblant à (*try .. catch*) de Java en utilisant les primitives C `setjmp` et `longjmp`. `setjmp` sauvegarde le contexte de pile et d'environnement afin de l'utiliser ultérieurement. `longjmp` restitue l'environnement sauvegardé lors du dernier appel de `setjmp`.

Cette implémentation a modifié le lanceur. La figure 5.10 présente la nouvelle version du lanceur de la figure 3.9. Les macros lignes 1–4 définissent le *try .. catch*. À la ligne 4,

```
1 #define TRY       switch( setjmp(ex_buf__) ){ case 0:
2 #define CATCH(x)  break; case x:
3 #define ETRY      }
4 #define THROW(x)  longjmp(ex_buf__, x)
5
6 int main() {
7   pathcrawler_init_streams();
8   while(1){
9     if(pathcrawler_finished())
10      return 0;
11    len = pathcrawler_read_length("str");
12    if(len==0)
13      str = pathcrawler_null();
14    else{
15      str = pathcrawler_alloc(len, sizeof(char));
16      pathcrawler_initialize(str, len);
17    }
18    TRY {
19      euchk(str);
20      pathcrawler_call_oracle();
21    }
22    CATCH ( ASSERT_EXCEPTION ){
23      pathcrawler_flush();
24    }
25    ETRY;
26    pathcrawler_free(str, len);
27  }
28 }
```

FIGURE 5.10 – Le lanceur de la figure 3.9 avec traitement des exceptions

le macro `THROW` correspond à l'appel de `lonjmp`. Les `TRY`, `CATCH` et `ETRY` sont insérés dans le code avant et après l'appel du programme sous test et de son oracle (lignes 18, 22 et 25 respectivement).

Le `THROW` est appelé par la fonction `error` présente dans les branches d'erreurs ajoutées (voir section 4.5.1). L'implémentation de la fonction `error` est présentée dans la figure 5.11. Le rôle de cette fonction est d'envoyer au moteur d'exécution symbolique (lignes 3–4) qu'une assertion a été violée en lui communiquant l'identifiant de cette dernière. Ensuite, on fait appel à `longjmp` par la macro `THROW` (ligne 5) pour revenir dans le lanceur et attendre les données du prochain cas de test.

Du côté du moteur d'exécution symbolique, l'appel de la fonction `error` ne doit pas être traité comme un appel de fonction ordinaire. Lors de l'étape d'analyse et d'instrumentation, cette fonction est traduite en PIF comme étant une instruction vide :

```
1 #define ASSERT_EXCEPTION  (1)   //code d'exception
2 int error(int ID) {
3   fprintf(pathcrawler_out_stream,"assert_violated('%d')\n", ID);
4   fflush(pathcrawler_out_stream);
5   THROW(ASSERT_EXCEPTION);
6 }
```

FIGURE 5.11 – Implémentation de la fonction `error`

```
block(+ID, []).
```

De plus, le moteur d'exécution symbolique doit comprendre le message reçu (`assert_violated`) à partir du lanceur et ensuite sauvegarder les données de ce test comme étant un contre-exemple. En réalité le message contient aussi des informations sur la localisation de ce bogue (nom du fichier et numéro de ligne dans le programme initial).

5.5 Calcul des ensembles couvrants

L'objectif des options avancées (*min* and *smart*) est de diminuer les appels très coûteux de l'analyse dynamique de l'option *each*. Ceci nécessite de calculer des ensembles couvrant toutes les alarmes d'une *couverture minimale des alarmes par slicing*.

En effet, nous avons prouvé dans le théorème 4.7.2 que toute *couverture minimale des alarmes par slicing* d'un ensemble d'alarmes A est définie par un ensemble de représentants des classes d'alarmes finales, et que les ensembles couvrants sont définis de manière unique. Un ensemble de représentants des classes d'alarmes finales peut être calculé en trois étapes comme suit :

(a) Une analyse de dépendance est utilisée pour calculer les dépendances pour chaque alarme. Nous utilisons le greffon d'analyse de dépendance distribué avec FRAMA-C, complété par une analyse de dépendance interprocédurale (le même calcul est utilisé par le *slicing*). Pour chaque alarme, cette analyse calcule la liste des instructions dont elle dépend. Ensuite, en filtrant toutes les instructions non menaçantes, on obtient la liste des alarmes dont elle dépend. Formellement parlant, pour chaque alarme a_i, on va trouver les alarmes a_j telles que $a_j \rightsquigarrow a_i$. Cette étape donne le graphe de dépendance $(A, \rightsquigarrow)$ qui correspond à la première étape de la figure 4.10.

(b) On identifie les alarmes finales dans le graphe de dépendance des alarmes $(A, \rightsquigarrow)$, en examinant les dépendances de chaque alarme avec les autres. Nous procédons ainsi :

1. Nous considérons toutes les alarmes comme candidates pour être des alarmes finales.
2. Nous choisissons une alarme a_1.
3. Nous choisissons un élément a_2 dans l'ensemble de ses dépendances ($a_2 \in deps(a_1)$).

4. Nous testons s'il n'y a pas de dépendance mutuelle entre l'alarme a_1 choisie à l'étape 2 et l'alarme a_2 choisie à l'étape 3, c'est-à-dire a_2 ne dépend pas de a_1 ($a_1 \notin deps(a_2)$), donc a_2 n'est pas une alarme finale et elle n'est plus considérée comme une candidate. Dans le cas contraire, où il y a une dépendance mutuelle, on garde les deux.
5. Nous répétons les étapes 3 et 4 pour tous les éléments dans l'ensemble des dépendances de l'alarme choisie.
6. Nous répétons les étapes 2, 3, 4 et 5 pour toutes les alarmes dans A.
7. Les candidates restantes sont les alarmes finales.

Pour le programme `eurocheck` présenté dans la figure 4.3, la figure 4.9 présente les dépendances entre les alarmes. Pour chaque alarme, les dépendances sont les suivantes :
- $deps(6_0) = \{6_0\}$
- $deps(13_0) = \{6_0, 13_0\}$
- $deps(15_0) = \{6_0, 15_0\}$
- $deps(17_0) = \{6_0, 15_0, 17_0\}$
- $deps(20_0) = \{6_0, 15_0, 17_0, 20_0\}$

Nous choisissons une alarme, par exemple 13_0. Ensuite nous choisissons l'élément 6_0 dans l'ensemble de ses dépendances ($deps(13_0)$). L'alarme 6_0 ne dépend pas de 13_0, donc il n'y a pas de dépendance mutuelle. On conclut que 6_0 n'est pas une alarme finale. Nous répétons les mêmes étapes 2, 3, 4 et 5 pour toutes les alarmes.

Pour le programme `eurocheck`, les alarmes finales sont 13_0 et 20_0.

(c) On choisit un ensemble de représentants des classes d'alarmes finales. Les représentants des classes d'alarmes finales peuvent être trouvés par une boucle de sélection de n'importe quelle alarme finale non encore marquée et marquant ses dépendances comme déjà représentées. Les étapes **(b)** et **(c)** sont représentées par l'étape "Sélectionner min" dans la figure 4.10. Pour le programme `eurocheck`, il n'y a pas de dépendances mutuelles entre les alarmes, donc l'ensemble des représentants des classes d'alarmes finales est $\{13_0, 20_0\}$.

Pour chaque élément a_i dans l'ensemble des représentants des classes d'alarmes finales, l'ensemble de dépendances $deps(a_i)$ est un ensemble couvrant. Pour l'exemple ci-dessus, les ensembles couvrants sont : $deps(13_0) = \{6_0, 13_0\}$ et $deps(20_0) = \{6_0, 15_0, 17_0, 20_0\}$.

Avec cette opération, on a réussi à diminuer les appels très coûteux de l'analyse dynamique de l'option *each* avec seulement un traitement supplémentaire de complexité polynomiale. Notons que l'étape **(a)** est déjà incluse dans l'option *each* où le *slicing* par rapport à chaque alarme appelle l'analyse de dépendance intra-procédurale et interprocédurale. Les étapes **(b)** et **(c)** ont seulement une complexité quadratique dans le nombre d'alarmes n.

5.6 SANTE CONTROLLER

Nous avons ajouté le greffon SANTE CONTROLLER à FRAMA-C. Comme son nom l'indique, ce nouveau greffon implémente le contrôleur de la méthode SANTE. Il appelle les différentes analyses dans le bon ordre. Aujourd'hui, il n'est pas distribué avec la version *open-source* de FRAMA-C car il nécessite une installation de PATHCRAWLER, outil propriétaire du CEA LIST.

SANTE CONTROLLER exécute les tâches suivantes

1. Il appelle l'analyse de valeurs.
2. Il récupère les alarmes signalées.
3. Il appelle ensuite le *slicing* en fonction du paramètre (*none*, *all*, *each*, *min*, *smart*) fourni par l'utilisateur.
4. Il analyse chaque programme simplifié avec PATHCRAWLER.
5. Il fournit le diagnostic final.

SANTE CONTROLLER assure aussi la circulation des informations entre les différentes analyses.

Un extrait de code source de l'implémentation de la fonction principale de SANTE CONTROLLERest présenté en annexe A.

5.6.1 État actuel

Notre prototype SANTE traite actuellement le langage C avec les tableaux, les pointeurs, les instructions, les conditionnelles, les boucles et les appels de fonctions. Il prend en entrée le code source du programme sous test, et la précondition de la fonction principale. Il produit en sortie les erreurs trouvées avec des cas de tests activant ces erreurs, et les menaces restantes qui sont de fausses alarmes (faux positifs) ou des erreurs qu'on n'a pas pu confirmer à cause de l'incomplétude du test.

Les sorties sont générées au format XML et sont visualisées à travers une interface web. La figure 5.12 montre les résultats pour l'exemple `eurocheck` de la figure 4.3. Pour chaque bogue détecté, il y a un lien qui permet de récupérer le contre-exemple menant à l'erreur.

La version actuelle du prototype n'inscrit pas les résultats obtenus dans la base de données de FRAMA-C. Cependant, nous envisageons de les ajouter prochainement.

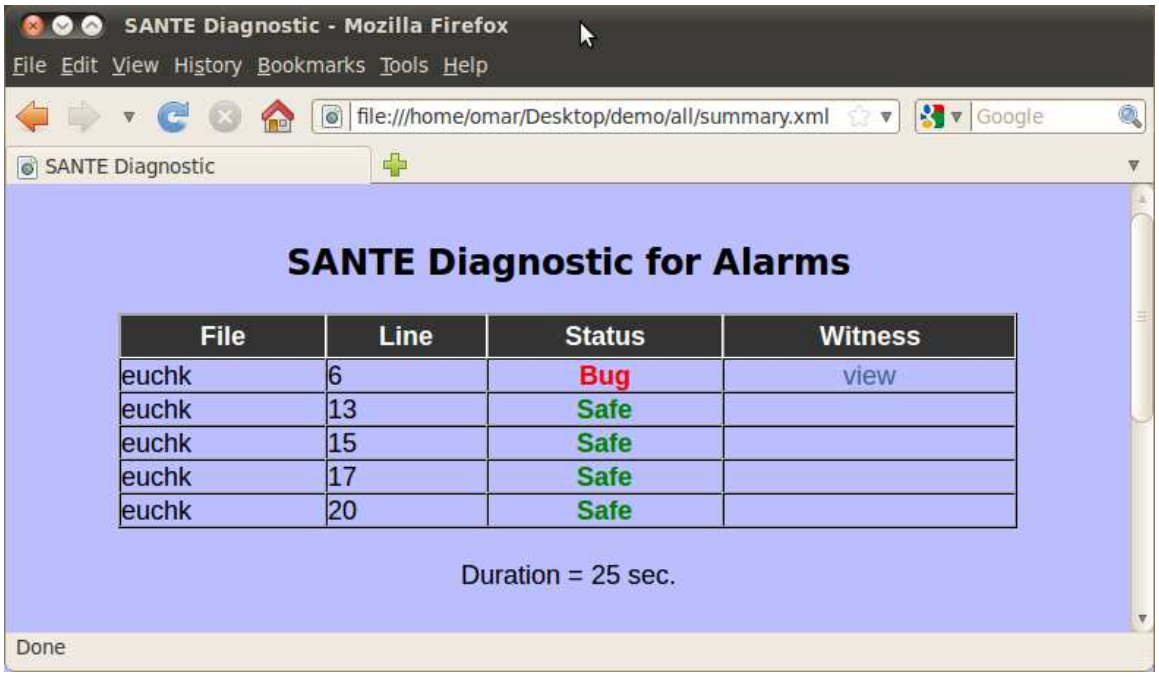


FIGURE 5.12 – Diagnostics de SANTE pour l'exemple `eurocheck` de la figure 4.3

Limitations

Les seules limites de la méthode SANTE sont celles des implémentations des outils choisis : analyse de valeurs, *slicing* et génération de tests dans FRAMA-C et PATHCRAWLER. Pour FRAMA-C, l'analyse de valeurs ne prend pas totalement en charge :

1. le type `long double`,
2. les fonctions avec un nombre variable d'arguments et
3. les boucles qui sont codées à partir des `goto` (l'analyse de ces boucles peut ne pas terminer).

Malgré ces limitations, les résultats de l'analyse de valeurs restent corrects.

Pour PATHCRAWLER, la version actuelle ne prend pas totalement en charge :

1. certaines sortes d'alias en entrée de la fonction sous test,
2. les fonctions et structures récursives,
3. les casts de pointeurs avec violations de type (*type-unsafe pointer casts*) et
4. le type `void*`.

5.6.2 Mode d'emploi

Comme nous l'avons déjà expliqué dans la section 5.6, SANTE CONTROLLER est implémenté comme un greffon de FRAMA-C. On peut le lancer par la commande `frama-c` en renseignant les options de SANTE. La commande est la suivante :

```
frama-c -sante fichier.c <option>
```

où `fichier.c` est le nom du fichier contenant le programme C à analyser. S'il y a plusieurs fichiers, on renseigne tous les noms des fichiers, séparés par un espace, comme dans l'exemple suivant :

```
frama-c -sante fichier1.c fichier2.c fichier3.c.
```

L'option `-main` permet de modifier le nom de la fonction qui sera le point d'entrée de l'analyse. Le point d'entrée par défaut est la fonction `main`.

Les options de l'analyse de SANTE sont :
- `-slice-all` pour lancer SANTE avec l'option *all* présentée dans la section 4.6.2.
- `-slice-each` pour lancer SANTE avec l'option *each* présentée dans la section 4.6.3.
- `-slice-min` pour lancer SANTE avec l'option *min* présentée dans la section 4.8.1.
- `-slice-smart` pour lancer SANTE avec l'option *smart* présentée dans la section 4.8.2.

Si aucune des options ci dessus n'est renseignée, SANTE CONTROLLER lance SANTE avec l'option *none* présentée dans la section 4.6.1.

L'option `-slevel` qui permet de paramétrer l'analyse de valeurs (voir section 3.4.3) est aussi applicable dans SANTE.

5.7 Synthèse

Ce chapitre a présenté le prototype SANTE CONTROLLER qui a permis de valider la méthode présentée dans ce mémoire. Nous avons présenté le greffon *PC Analyzer* développé pour relier l'outil PATHCRAWLER à la plate-forme FRAMA-C. L'article [KBR+12] présente la nouvelle version de PATHCRAWLER qui utilise ce greffon. Ensuite nous avons adapté PATHCRAWLER afin de traiter les annotations ACSL, en particulier les assertions. Nous avons introduit un mécanisme de traduction des conditions d'erreurs ACSL en conditions C exécutables. Nous avons implémenté un mécanisme d'exceptions dans PATHCRAWLER. Nous avons implanté le calcul des ensembles couvrants d'une couverture minimale par *slicing*. Une démonstration de l'outil SANTE a été faite à la conférence AFADL 2010 [Che10].

Le chapitre suivant présente les expérimentations effectuées à partir de cette implémentation et les résultats obtenus permettant d'évaluer les idées énoncées dans ce travail de thèse.

Chapitre 6

Expérimentations

Sommaire

6.1 Introduction

Dans ce chapitre, nous présentons des résultats d'expérimentations dans lesquelles nous avons utilisé le logiciel SANTE qui implémente la méthode de vérification que nous avons définie. L'objectif de ce travail expérimental est de valider l'approche sur un ensemble d'exemples bien choisis et de mettre en évidence ses limites. Nous essayons d'adopter une approche expérimentale habituelle en recherche scientifique, en fixant tout d'abord dans la partie 6.2 les hypothèses que nous faisons sur l'apport de la méthode. Une hypothèse énonce un résultat attendu, qui induit une question générique : est-ce que les résultats obtenus expérimentalement confirment, infirment ou ne permettent pas de conclure que l'hypothèse est vraie ?

Dans la partie 6.3, nous définissons les conditions expérimentales en présentant les exemples de programmes utilisés et nous justifions leur choix par les objectifs de l'expérience et les limites de l'implémentation. Nous présentons également le matériel utilisé et les constantes expérimentales pour permettre la reproductibilité des expériences. Dans la partie 6.4, nous présentons les autres méthodes avec lesquelles on compare les résultats obtenus en utilisant SANTE. Ces autres méthodes sont l'analyse de valeurs seule et son implantation dans FRAMA-C et l'analyse dynamique seule, appelée *all-threats DA*, que nous avons implémentée en utilisant PATHCRAWLER dans des conditions particulières qui sont de rechercher toutes les menaces potentielles. Dans la partie 6.5, nous présentons les critères mesurés au cours des expériences. Ces critères sont choisis afin de pouvoir répondre aux questions induites par les hypothèses de la partie 6.2.

Nous divisons les expérimentations en trois parties. La partie 6.6 présente l'apport de la combinaison par rapport à chaque technique utilisée séparément en comparant SANTE utilisant l'option *none* avec l'analyse de valeurs seule et l'analyse dynamique seule représentée par la méthode *all-threats DA*. La partie 6.7 présente l'apport du *slicing* dans chaque utilisation (*all*, *each*, *min* et *smart*) par rapport à SANTE sans *slicing* (*none*) et compare les options les unes avec les autres. La partie 6.8 présente l'effet du *slicing* sur la taille des programmes, sur la longueur et sur le nombre de chemins analysés. Dans chaque partie, nous présentons et analysons les résultats et nous fournissons une synthèse.

Pour terminer, la partie 6.9 conclut par des réponses aux questions sous-jacentes aux hypothèses et par l'énoncé des limites de notre méthode et de nos expérimentations. Enfin, nous dégageons des perspectives de ces limites, qui seront détaillées dans le chapitre 7.

6.2 Hypothèses et objectifs des expérimentations

On identifie les hypothèses des expérimentations comme suit :

$\mathcal{H}_1$ – La combinaison de l'analyse de valeurs et de l'analyse dynamique apporte de meilleurs résultats de classification (laisse moins d'alarmes non classées) dans un délai fixé que l'analyse de valeurs seule ou l'analyse dynamique seule.

N°	Origine	Fonction principale	Taille	Nb. des menaces potentielles	Bogues connus
1	Apache	escape_absolute_uri (simplified)	33	8	1
2	Apache	escape_absolute_uri (full)	97	16	1
3	Spam Assassin	message_write	55	17	2
4	Apache	get_tag	696	12	3
5	QuickSort	partition	50	8	1
6	libgd	gdImageStringFTEx	705	15	1
7	polygon	main	202	29	2
8	rawcaudio	adpcm_decoder	365	10	0
9	eurocheck	main	154	19	1
Total				134	12

FIGURE 6.1 – Informations sur les exemples d'expérimentation

$\mathcal{H}_2$ – Les différentes manières de faire du *slicing* améliorent encore les résultats. Elles permettent de classer plus d'alarmes ou d'améliorer le temps d'exécution de la phase de génération de tests.

$\mathcal{H}_3$ – L'option *smart* est la meilleure méthode sur les deux critères suivants : nombre d'alarmes classées et durée d'analyse.

$\mathcal{H}_4$ – Le *slicing* permet d'obtenir des informations plus précises sur les erreurs et les alarmes détectées.

6.3 Conditions expérimentales

Nous utilisons neuf exemples, énumérés dans la figure 6.1. Les colonnes de la figure 6.1 présentent respectivement le numéro de l'exemple, son origine, le nom de la fonction analysée, la taille de chaque exemple en nombre de lignes de code, le nombre total de menaces potentielles et le nombre de bogues connus parmi ces menaces.

Ces neufs exemples sont extraits du monde réel. Leurs erreurs sont connues car elles ont déjà été détectées. Les exemples sont choisis aléatoirement, en prenant les limites d'implémentation en compte. D'autres exemples avec des résultats similaires ont été écartés par souci de clarté. Ainsi, chacun des neuf exemples restants représente une catégorie d'exemples, comme on le verra dans les parties 6.6, 6.7 et 6.8. Toutes les erreurs sont de type accès hors limite ou pointeur invalide. Les exemples 1, 2, 3, 4 et 6 proviennent du banc de test *Verisec C analysis benchmark* [KHCL07]. L'exemple 2 est un composant Apache. Il sert à parser un identificateur de ressource (URI ou *Uniform Resource Identifier*). L'exemple 1 est une version simplifiée de l'exemple 2. L'exemple 3 est un programme *open-source* [Ray98] qui sert à filtrer les courriers indésirables (*spam*). L'exemple 4 est un composant Apache. Il sert à parser un document HTML. L'exemple 5 provient de [Bal04].

C'est l'exemple classique utilisé par le tri rapide qui permet de trier un tableau par partition. L'exemple 6 est un programme *open-source* qui permet de créer des images et des graphes dynamiquement. L'exemple 7 est un programme *open-source*. Il provient de [Cod11] et sert à calculer la surface d'un polygone convexe à partir des coordonnées de ses sommets. L'exemple 8 provient du banc de test *Mediabench* [LPMS97]. Il sert à coder un fichier audio de format PCM en format ADPCM et l'inverse. L'exemple 9 est un programme *open-source* [DOS99]. Il provient de [Rut11] et sert à vérifier si le numéro de série d'un billet de banque en euros est valide ou non.

Les expériences ont été menées sur un processeur Intel Duo 1,66 GHz avec une carte mémoire de 1 Go (RAM) avec un *timeout* de 10 minutes.

6.4 Méthodes comparées

Une grande partie des outils cités dans le chapitre 2 ne sont pas disponibles et nous ne pouvons pas les utiliser dans nos expérimentations. C'est pourquoi nous n'avons pas comparé SANTE avec eux. Une méthode similaire à celle de l'outil EXE présenté dans la partie 2.3 était implantable avec un effort raisonnable en se servant d'outils existant déjà au laboratoire de sûreté des logiciels au CEA. Pour cette raison, nous avons choisi comme méthode de référence une méthode d'analyse dynamique *all-threats DA* (*all-threats dynamic analysis*). Nous avons développé cette méthode, qui guide la génération de tests avec la liste exhaustive des alarmes, pour toutes les instructions a priori menaçantes, et sans analyse de leur contexte.

Une instruction a priori menaçante est une instruction qui présente un risque potentiel d'erreur à l'exécution comme par exemple une division, un accès tableau ou accès pointeur. Autrement dit, toute instruction qui effectue une division par une expression ou un accès dans un tableau est potentiellement menaçante. Nous ignorons le contexte dont on pourrait peut-être déduire que c'est une fausse menace. À chaque fois qu'il y a une expression de division, *all-threats DA* va essayer de trouver un cas de test provoquant la division par zéro. De même, à chaque fois qu'il y a un accès mémoire, *all-threats DA* essaie de trouver un accès invalide.

all-threats DA utilise PATHCRAWLER en mode "génération de test guidée par les alarmes" (*alarm-guided test generation*) comme SANTE. La différence est que SANTE considère uniquement les alarmes signalées par l'analyse de valeurs tandis que *all-threats DA* considère toutes les instructions potentiellement menaçantes comme des alarmes.

On considère également, comme une autre référence la méthode *VA*, qui consiste à utiliser l'analyse de valeurs seule.

Nous comparons *VA* et *all-threats DA* avec les méthodes *none*, *all*, *each*, *min* et *smart* qui correspondent aux options fournies par SANTE. Nous comparons également les options

de SANTE entre elles.

6.5 Critères de comparaison

Les critères mesurés sont les suivants :

$\mathcal{C}_1$ – Le nombre de menaces classées safe, c'est-à-dire les menaces dont l'état d'erreur est prouvé inatteignable par l'analyse de valeurs ou par la génération de tests exhaustive de tous les chemins dans le cas où elle termine. Ce nombre est indiqué dans les colonnes '✓' des tableaux de résultats.

$\mathcal{C}_2$ – Le nombre d'alarmes restant non classées, indiqué dans les colonnes '?'.

$\mathcal{C}_3$ – Le nombre de bogues détectés (menaces classées bug), indiqué dans les colonnes '↯'.

$\mathcal{C}_4$ – La durée du processus complet, indiquée dans les colonnes 'temps'.

$\mathcal{C}_5$ – Le nombre de chemins traités (faisables et infaisables) par la génération de tests, indiqué dans les colonnes 'chemins'.

$\mathcal{C}_6$ – La longueur moyenne des chemins parcourus.

$\mathcal{C}_7$ – La longueur moyenne des contre-exemples.

$\mathcal{C}_8$ – La taille du programme en nombre de lignes de code.

Dans les colonnes 'temps', 'TO' indique la présence d'un *time / space out*. Ces critères vont permettre d'évaluer si les hypothèses, identifiées dans la partie 6.1, sont satisfaites ou non. Les hypothèses $\mathcal{H}_1 - \mathcal{H}_3$ seront quantifiée à l'aide des critères $\mathcal{C}_1 - \mathcal{C}_5$. Les critères $\mathcal{C}_6 - \mathcal{C}_8$ seront utilisés pour quantifier l'hypothèse $\mathcal{H}_4$.

Nous sommes maintenant prêts à présenter les résultats des expériences sur chaque exemple, dans les parties 6.6, 6.7 et 6.8, en mesurant essentiellement le nombre d'alarmes non classées, qui quantifie l'imprécision de la méthode, et le temps total d'analyse. Nous présentons également des résultats en moyenne évalués sur l'ensemble d'exemples de programmes, puis nous faisons un bilan relativement aux hypothèses.

6.6 Apport de la combinaison par rapport à chaque technique seule

Dans cette partie, nous comparons l'option *none* de la méthode SANTE, qui combine l'analyse statique et la génération de tests structurels, sans simplification par le *slicing*, avec l'analyse de valeurs et avec la technique d'analyse dynamique *all-threats DA* (présentée dans la partie 6.4) utilisées indépendamment. Dans cette comparaison, nous allons mesurer les critères $\mathcal{C}_1 - \mathcal{C}_5$.

N°	Analyse de valeurs			*all-threats* DA			SANTE *none*		
	✓	?	↯	✓	?	↯	✓	?	↯
1	4	**4**	0	7	**0**	1	7	**0**	**1**
2	11	**5**	0	15	**0**	1	15	**0**	**1**
3	2	**15**	**0**	0	**17**	0	15	**0**	**2**
4	0	**12**	**0**	0	**12**	**0**	0	**12**	**0**
5	4	**4**	0	7	**0**	1	7	**0**	1
6	3	**12**	0	0	**14**	1	3	**11**	1
7	19	**10**	0	27	**0**	2	27	**0**	2
8	8	**2**	0	0	**10**	0	8	**2**	0
9	14	**5**	0	18	**0**	1	18	**0**	1
Total	65	69	0	74	53	7	100	25	9

FIGURE 6.2 – Résultats de classification par *VA*, *all-threats DA* et SANTE *none* avec le critère "tous les chemins"

6.6.1 Classification avec la couverture "tous les chemins"

La figure 6.2 compare l'option *none* de SANTE avec les deux autres techniques, avec le critère "tous les chemins" pour PATHCRAWLER. On note que, à l'exception de l'exemple 4, tous les bogues connus sont détectés avec SANTE *none*. Les deux analyses comparatives suivantes exploitent de manière détaillée ces premiers résultats.

SANTE vs analyse de valeurs.

La figure 6.2 montre que, dans la plupart des cas (exemples 1, 2, 3, 5, 6, 7, 8 et 9), l'analyse de valeurs seule réduit le nombre de menaces potentielles et prouve que certaines d'entre elles sont safe, mais génère une forte proportion d'alarmes, plus de 50% dans les exemples 1, 3, 4, 5 et 6. Nous voyons dans la même figure que, dans sept cas sur huit, SANTE confirme certaines alarmes comme de vrais bogues en fournissant un cas activant chaque bogue. Nous voyons également que SANTE laisse beaucoup moins d'alarmes avec le statut unknown en prouvant que certaines menaces sont fausses. Dans cinq cas, toutes les alarmes sont classées vraies ou fausses. Sur l'ensemble des neuf exemples, SANTE classe 44 alarmes, parmi les 69 alarmes que l'analyse de valeurs n'a pas classées.

SANTE vs *all-threats DA*.

SANTE détecte le même nombre d'erreurs que *all-threats DA* pour tous les programmes, sauf pour le 3, où il en trouve deux de plus. SANTE classe beaucoup plus de menaces, en éliminant des menaces dans le cas des exemples 3, 6 et 8. Sur les neuf exemples, il élimine 28 alarmes parmi les 53 que *all-threats DA* n'a pas classées. Notons que SANTE *none*

N°	*all-threats* DA		SANTE *none*	
	chemins	temps	chemins	temps
1	3602	22s	2454	14s
2	3876	20s	2694	10s
3	TO		37967	8min 43s
4	TO		TO	
5	18978	1min 56s	15250	1min 12s
6	TO		TO	
7	23502	5min 33s	12603	1min 31s
8	TO		TO	
9	1186	25s	878	18s

FIGURE 6.3 – Temps consommé par *all-threats DA* et SANTE *none* avec le critère "tous les chemins"

trouve toutes les erreurs connues, sauf pour l'exemple 4, et que *all-threats DA* les trouve toutes sauf pour les exemples 3 et 4.

6.6.2 Durée d'analyse et chemins traités avec la couverture "tous les chemins"

La figure 6.3 compare les temps d'exécution entre les différentes analyses et le nombre de chemins traités par la génération de tests pour les techniques *all-threats DA* et SANTE, avec le critère "tous les chemins" pour PATHCRAWLER.

La génération de tests guidée par les alarmes dans SANTE traite seulement les alarmes maintenues par l'analyse de valeurs alors que *all-threats DA* considère naïvement toutes les menaces potentielles. Ainsi, la génération de test dans SANTE considère moins de chemins. Pour les exemples 1, 2, 5, 7 et 9, SANTE détecte le même nombre d'erreurs que *all-threats DA* en moins de temps. Pour l'exemple 3, le gain de temps permet à SANTE d'éviter le dépassement de temps limite et aussi de classer deux erreurs de plus. Pour les cas 4, 6 et 8, le gain de temps est insuffisant pour classer toutes les alarmes.

Dans l'exemple 9, *all-threats DA* analyse 19 alarmes (voir figure 6.2 exemple 9), et il faut 25 secondes pour trouver un bogue et prouver que les états d'erreur sont inatteignables pour les 18 autres menaces (voir figure 6.3 exemple 9), tandis que l'analyse dynamique dans SANTE analyse 5 alarmes, parce que 14 ont été déjà prouvées safe par l'analyse de valeurs. SANTE *none* prend 18 secondes pour classer toutes ces alarmes. (voir figure 6.3 exemple 9). La diminution du temps est la plus importante en proportion dans l'exemple 7 car SANTE *none* analyse 10 alarmes alors que *all-threats DA* en analyse 27.

N°	k	*all-threats DA*			SANTE *none*		
		✓	?	↯	✓	?	↯
1	3	0	**7**	1	4	**3**	**1**
2	10	0	**15**	1	11	**4**	**1**
3	3	0	**15**	2	2	**13**	**2**
4	2	0	**9**	3	0	**9**	**3**
5	2	0	**7**	1	4	**3**	**1**
6	3	0	**14**	1	3	**11**	**1**
7	2	0	**27**	2	19	**8**	**2**
8	2	0	**10**	0	8	**2**	**0**
9	2	0	**18**	1	14	**4**	**1**
Total		0	122	12	65	57	12

FIGURE 6.4 – Résultats de classification par *all-threats DA* et SANTE *none* avec le critère "tous les k-chemins"

SANTE *none* termine dans le cas de l'exemple 3, où *all-threats DA*, fait un *time / space out*. Dans le pire des cas, où l'analyse de valeurs ne filtre aucune menace, SANTE *none* pourrait prendre autant de temps, voire un peu plus, que *all-threats DA* puisqu'elle effectue l'analyse de valeurs en plus. Mais aucun de nos exemples n'illustre cette situation. C'est peut-être le cas de l'exemple 4 où l'analyse de valeurs ne classe aucune alarme. Mais le dépassement de temps dans les 2 cas ne permet pas d'observer cette situation.

6.6.3 Classification avec la couverture "tous les k-chemins"

Nous répétons les mêmes expérimentations avec le critère de test partiel "tous les k-chemins" (*k-path*). La stratégie "tous les k-chemins" est donnée pour le k minimal permettant de détecter tous les bogues connus avec SANTE.

Le critère "tous les k-chemins" n'affectant pas l'analyse de valeurs, la figure 6.4 ne compare pas SANTE avec l'analyse de valeurs, mais seulement avec *all-threats DA*.

SANTE détecte le même nombre de bogues que *all-threats DA* sur tous les exemples et laisse beaucoup moins d'alarmes avec le statut unknown (Exemples 1, 2, 3, 5, 6, 7, 8 et 9). Mais notons que les fausses alarmes classées safe par SANTE l'ont toutes été par l'analyse de valeurs. En effet, le parcours avec le critère de couverture "tous les k-chemins", étant partiel, il ne peut pas garantir l'absence d'erreurs et ne permet de classer aucune alarme safe. Donc la phase d'analyse dynamique de SANTE n'apporte pas d'amélioration autre que celle de *VA* concernant l'élimination de fausses alarmes. Dans le pire des cas, lorsque l'analyse de valeurs dans SANTE ne filtre aucune menace, SANTE donne les mêmes résultats que *all-threats DA* (Exemple 4). Notons qu'il est normal que SANTE *none* et *all-*

N°	k	*all-threats DA*		SANTE *none*	
		chemins	temps	chemins	temps
1	3	71	<1s	45	<1s
2	10	417	1s	325	1s
3	3	30977	9min 18s	18874	3min 35s
4	2	36690	TO	36690	TO
5	2	4509	25s	3319	18s
6	3	19806	9s	15544	8s
7	2	195	1s	88	1s
8	2	15	<1s	7	<1s
9	2	60	1s	18	<1s

FIGURE 6.5 – Temps consommé et nombre de chemins parcourus par *all-threats DA* et SANTE *none* avec le critère "tous les k-chemins"

threats DA trouvent les mêmes erreurs avec la même valeur de k car elles activent toutes les deux PATHCRAWLER dans les mêmes conditions. Mais SANTE *none* pourrait détecter plus de bogues sur un exemple où *all-threats DA* ferait un *time / space out* et SANTE *none* terminerait, car *all-threats DA* doit analyser plus d'alarmes, donc plus de chemins, donc prend plus de temps. Mais aucun exemple n'illustre cette situation.

6.6.4 Durée d'analyse et chemins traités avec la couverture "tous les k-chemins"

La figure 6.5 compare les temps d'exécution entre les différentes analyses et le nombre de chemins traités par la génération de tests pour les techniques *all-threats DA* et SANTE avec le critère "tous les k-chemins" pour PATHCRAWLER.

La génération de tests dans SANTE considère moins de chemins, comme nous le voyons sur les exemples 1, 2, 3 et 5. SANTE détecte le même nombre de bogues que *all-threats DA* en moins de temps. Dans le pire des cas, lorsque l'analyse de valeurs dans SANTE ne filtre aucune menace, SANTE peut prendre autant de temps que *all-threats DA* (Exemple 4), voire un peu plus.

Une difficulté d'application de SANTE avec le critère "tous les k-chemins" est liée au choix de la valeur de k. Dans nos expérimentations, la valeur minimale de k permettant de détecter tous les bogues connus est trouvé séquentiellement, ce qui n'est pas efficace. De plus, dans une application réelle, ne connaissant pas le nombre de bogues connus, nous ne pourrons plus utiliser cette information comme critère d'arrêt. L'impossibilité de connaître la valeur de k qui permettra de détecter toutes les erreurs reste une limitation importante de la méthode SANTE avec le critère "tous les k-chemins".

Enfin, notons que la stratégie "tous les k-chemins" peut s'avérer plus efficiente que la stratégie "tous les chemins", comme l'illustre l'exemple 4 où, malgré le dépassement de temps limite, les trois erreurs connues sont trouvées avec $k = 2$ alors que la stratégie "tous les chemins" n'en trouve aucune.

6.6.5 Bilan de cette expérimentation

Les principales conclusions de cette partie sont :

1. L'outil SANTE combinant l'analyse de valeurs et PATHCRAWLER classe plus de menaces que chacun de ces outils utilisés séparément,

2. SANTE est plus efficace (en temps) qu'un outil de test structurel seul,

3. La génération de test guidée par la liste exhaustive des alarmes pour toutes les instructions potentiellement menaçantes examine plus de chemins infaisables que SANTE et prend donc plus de temps. Parfois elle peut faire un *time / space out*.

4. Avec le critère de test partiel "tous les k-chemins" (*k-path*), SANTE *none* et *all-threats DA* détectent le même nombre de bogues mais SANTE *none* laisse moins d'alarmes non classées, grâce au travail préalable de l'analyse de valeurs.

5. Avec le critère "tous les chemins", SANTE *none* et *all-threats DA* classent plus d'alarmes qu'avec le critère "tous les k-chemins", sauf quand il y a un *time / space out*.

6. Avec le critère "tous les k-chemins", SANTE *none* et *all-threats DA* ont réussi à détecter tous les bogues dans une durée beaucoup plus courte qu'avec le critère "tous les chemins". Mais la définition d'une méthode pour choisir k reste un travail à faire avec pour objectif de trouver un compromis raisonnable et acceptable entre la précision de la classification des alarmes et le temps d'analyse du programme.

7. Malgré l'apport de la combinaison par rapport à chaque technique utilisée indépendamment, les expérimentations montrent aussi que SANTE *none* peut faire des dépassements de temps limite et laisser beaucoup d'alarmes non classées.

N°	SANTE *none*			SANTE *all*			SANTE *each*			SANTE *min*			SANTE *smart*		
	✓	?	↯	✓	?	↯	✓	?	↯	✓	?	↯	✓	?	↯
1	7	0	1	7	0	1	7	0	1	7	0	1	7	0	1
2	15	0	1	15	0	1	15	0	1	15	0	1	15	0	1
3	15	0	2	15	0	2	15	0	2	15	0	2	15	0	2
4	0	12	0	0	9	3	4	5	3	0	9	3	4	5	3
5	7	0	1	7	0	1	7	0	1	7	0	1	7	0	1
6	3	11	1	3	11	1	14	0	1	14	0	1	14	0	1
7	27	0	2	27	0	2	27	0	2	27	0	2	27	0	2
8	8	2	0	8	2	0	9	1	0	8	2	0	9	1	0
9	18	0	1	18	0	1	18	0	1	18	0	1	18	0	1
Total	100	25	9	100	22	12	116	6	12	111	11	12	116	6	12

FIGURE 6.6 – Résultats de classification avec les différentes options de SANTE

6.7 Apport du *slicing* en classification et temps d'analyse

Dans cette partie, nous présentons des expérimentations avec les différentes options de SANTE. Afin de montrer l'apport de chaque utilisation du *slicing*, nous comparons les options les unes avec les autres et avec la méthode SANTE *none* et un parcours "tous les chemins", dont on a présenté les résultats dans la partie précédente. Dans cette comparaison, nous allons mesurer les critères $\mathcal{C}_1 - \mathcal{C}_4$. Nous comparons les résultats de classification des différentes méthodes dans la partie 6.7.1 et les temps d'analyse dans la partie 6.7.2.

6.7.1 Comparaison des résultats de classification des différentes méthodes

La figure 6.6 présente les résultats de la classification des alarmes pour chaque technique et pour chaque exemple. On note que tous les bogues connus sont détectés avec les méthodes utilisant le *slicing*. Ce n'est pas le cas avec SANTE *none* sur l'exemple 4.

On peut noter que, tous les exemples, SANTE *smart* et SANTE *each* sont les deux méthodes qui classent le plus d'alarmes. Au total, il reste 6 alarmes non classées. Ensuite c'est SANTE *min* qui en laisse 11, puis SANTE *all* qui en laisse 22 et enfin SANTE *none* qui laisse 25 alarmes non classées.

SANTE ***none*** **vs** SANTE ***all.***
La génération de tests dans SANTE *all* analyse un programme simplifié contenant toutes les alarmes, donc elle considère moins de chemins que SANTE *none*. SANTE *all* donne au

minimum les mêmes résultats que SANTE *none*. Il a plus de chance de détecter des bogues que SANTE *none* car SANTE *all* parcourt moins de chemins. C'est ce qui se produit dans le cas de l'exemple 4 où malgré un dépassement de temps limite dans les deux cas, SANTE *all* trouve les trois erreurs alors que SANTE *none* n'en trouve aucune. SANTE *all* pourrait terminer et classer des alarmes dans des cas ou SANTE *none* ferait un *time / space out*.

Les contre-exemples reportés par SANTE *all* sont illustrés sur un programme simplifié ce qui rend l'investigation de la source d'erreur plus facile.

SANTE ***all*** **vs** SANTE ***each.***
Dans SANTE *each*, la *slice* minimale pour chaque alarme est analysée séparément par la génération de tests. SANTE *each* laisse moins d'alarmes avec le statut unknown (voir figure 6.6 exemples 4, 6 et 8). SANTE *each* termine dans certains cas où SANTE *all* fait un *time / space out* (voir figure 6.7 exemples 4, 6 et 8). Il classe plus d'alarmes comme dans l'exemple 8, où l'analyse dynamique analyse deux *slices*, l'une d'entre elles fait un *time / space out* et l'autre termine et classe l'alarme cible comme safe. Les contres-exemples reportés sont illustrés sur des programmes plus simples, voire minimaux, ce qui rend l'investigation de la source d'erreur encore plus facile qu'avec SANTE *all*.

SANTE ***all, each*** **vs** SANTE ***min.***
Dans SANTE *min*, l'analyse dynamique sur certaines *slices* peut terminer dans certains cas où elle fait un *time / space out* dans SANTE *all* et donc elle peut classer plus d'alarmes (voir figure 6.6 exemple 6). Sur l'ensemble des exemples, SANTE *min* classe 11 alarmes de plus que SANTE *all*. Toutefois, l'analyse dynamique dans SANTE *min* peut aussi faire un *time / space out* alors que SANTE *each* termine sur quelques *slices* et classe plus d'alarmes (voir figure 6.6 exemples 4 et 8). Sur l'ensemble des exemples, SANTE *each* classe 5 alarmes de plus que SANTE *min*. En effet l'analyse dynamique sur la *slice* de taille minimale pour une alarme a la plus grande chance de la classer.

SANTE ***each, min*** **vs** SANTE ***smart.***
En l'absence de *time / space out*, SANTE *smart* se comporte exactement comme SANTE *min* (voir figure 6.6 exemples 6, 7 et 9). Si un *time / space out* est rencontré, SANTE *smart* analyse des *slices* de plus en plus petites et ne s'arrête que quand il ne peut plus classer d'alarmes. Ainsi, il peut classer plus d'alarmes que SANTE *min* (voir figure 6.6 exemples 4 et 8). Sur l'ensemble des exemples SANTE *smart* classe 5 alarmes de plus que SANTE *min*. SANTE *smart* trouve les mêmes résultats que SANTE *each*. En effet, il peut aller jusqu'à analyser la *slice* minimale pour une alarme, s'il n'a pas réussi à la classer avec les *slices* de plus grande taille.

N°	SANTE *none*	SANTE *all*	SANTE *each*	SANTE *min*		SANTE *smart*		
				temps	*slices*	temps	*slices*	itérations
1	14s	11s	54s	11s	1	11s	1	1
2	10s	7s	15s	7s	1	7s	1	1
3	8min 43s	8min 43s	1h 5min 16s	17min 13s	4	17min 13s	4	1
4	TO	TO	3min 24s + 5 TO	1 TO	1	54s + 1 TO	2	2
5	1min 12s	1min 12s	4min 48s	1min 12s	1	1min 12s	1	1
6	TO	TO	1h 32min 52s	32min 16s	4	32min 16s	4	1
7	1min 31s	1min 20s	7s	7s	9	7s	9	1
8	TO	TO	5s + 1 TO	1 TO	1	5s + 1 TO	2	2
9	18s	7s	13s	6s	2	6s	2	1

FIGURE 6.7 – Temps consommé par les différentes options de SANTE

6.7.2 Comparaison des temps d'analyse des différentes méthodes

La figure 6.7 présente les temps d'analyse pour chaque exemple. Par exemple, 3min 24s + 5TO sur l'exemple 4 signifie que sur l'analyse des 12 *slices*, 5 terminent en *time / space out* et les sept autres prennent 3 minutes et 24 secondes pour classer leurs alarmes.

Pour les colonnes *min* et *smart*, la figure 6.7 indique le nombre de *slices* analysées. Pour *all*, il n'y en a qu'une et pour *each*, il y en a autant que d'alarmes non classées par l'analyse de valeurs indiquées dans la figure 6.2. De plus, la figure 6.7 indique le nombre d'itérations de l'algorithme *smart*.

SANTE ***none*** **vs** SANTE ***all.***
La génération de tests dans SANTE *all* analyse un programme simplifié contenant toutes les alarmes. Donc il considère moins de chemins et donne les mêmes résultats en moins de temps (voir figure 6.7 exemple 9). SANTE *all* pourrait terminer dans certains cas où SANTE *none* ferait un *time / space out* et donc SANTE *all* pourrait classer plus d'alarmes.

Dans le pire des cas, lorsque le *slicing* ne simplifie pas énormément le programme, SANTE *all* prend presque autant de temps que SANTE *none* (voir figure 6.7 exemple 7).

SANTE ***all*** **vs** SANTE ***each.***
Dans SANTE *each*, une *slice* est analysée dynamiquement pour chaque alarme. Le temps complet est la somme des temps nécessaires pour analyser toutes les *slices*. Par exemple,

pour le programme 4, l'analyse de valeurs signale 12 alarmes. Avec l'option *each*, la méthode produit 12 *slices*. Chacune des *slices* est analysée séparément par l'analyse dynamique. L'analyse dynamique de 5 de ces *slices* fait des *time / space out* alors que les 7 restantes terminent et classent les 7 alarmes correspondantes. La durée totale requise pour analyser le programme initial avec l'analyse de valeurs, puis produire les 12 *slices*, ensuite analyser ces 7 *slices* et classer ces 7 alarmes par l'analyse dynamique est de 3 minutes 24 secondes.

SANTE *each* peut être plus lent que SANTE *all* (voir figure 6.7 exemple 9). De plus, l'analyse dynamique de certaines *slices* avec SANTE *each* peut ne pas donner de nouvelles informations (voir parties 4.6 et 4.8) et c'est alors une perte de temps. C'est le cas avec l'exemple 9 de la figure 6.7. Pour cet exemple, on peut dire que SANTE *all* est une meilleure méthode que SANTE *each*. Elle classe autant d'alarmes en moins de temps.

Dans certains cas SANTE *each* peut être plus rapide que SANTE *all* et laisser autant d'alarmes, comme dans le cas de l'exemple 7, où l'analyse de valeurs signale 10 alarmes qui sont toutes indépendantes. L'analyse dynamique analyse 10 *slices* dont les tailles sont largement inférieures à celles du programme initial. Le nombre de chemins traités (faisables et infaisables) est beaucoup plus faible et surtout la taille des chemins traités est beaucoup plus petite.

SANTE ***all*, *each*** **vs** SANTE ***min*.**
Dans SANTE *min*, l'analyse dynamique analyse moins de programmes que dans SANTE *each*, elle est donc plus rapide (voir figure 6.7 exemples 6 et 9). De plus, l'analyse dynamique des *slices* dans SANTE *min* n'est jamais une perte de temps.

SANTE *min* peut terminer dans des cas où SANTE *all* fait un *time / space out* (voir figure 6.7 exemple 6).

Néanmoins, SANTE *min* peut faire un *time / space out* dans des cas où SANTE *each* termine sur certaines *slices* et classe certaines alarmes (voir figure 6.7 exemples 4 et 8). En effet l'analyse dynamique sur la *slice* de plus petite taille pour une seule alarme a la plus grande chance de la classer.

SANTE ***each*, *min*** **vs** SANTE ***smart*.**
SANTE *smart* peut prendre plus de temps que SANTE *min*, mais offre de meilleures réponses dans certains cas (voir figure 6.7 exemples 4 et 8).

SANTE *smart* est plus rapide que SANTE *each* (voir figure 6.7 exemples 6 et 9). De plus, l'analyse dynamique des *slices* dans SANTE *smart* n'est jamais une perte de temps au sens où c'est l'option qui a le meilleur taux de classification, taux identique à celui de SANTE *each* (voir figure 6.7 exemples 6, 4 et 9).

Par exemple, pour l'exemple 4, l'analyse dynamique dans SANTE *each* analyse 12 *slices* et fait 5 *time / space out*. Dans SANTE *min*, l'analyse dynamique analyse une *slice* seulement et fait un *time / space out*. SANTE *smart* a besoin de deux itérations pour conclure. Dans la première itération, l'analyse dynamique analyse une *slice* contenant toutes les 12 alarmes. Elle classe 3 alarmes comme bug et fait un *time / space out*. Ensuite elle conclut dans la deuxième itération où elle analyse une *slice* contenant 4 alarmes en 54s (5 alarmes finales mutuellement dépendantes ont été retirées). SANTE *smart* trouve les mêmes résultats que SANTE *each* après seulement un *time / space out* alors que SANTE *each* en effectue cinq.

6.7.3 Bilan de cette expérimentation

Les principales conclusions de cette partie sont :

1. Le *slicing* des programmes permet d'améliorer le nombre d'alarmes classées et le temps d'exécution de la phase de génération de tests, en l'appliquant sur des programmes simplifiés.
2. Les options *each* et *smart* sont les options qui classent le plus d'alarmes. Mais l'option *each* peut faire des appels inutiles à l'analyse dynamique. Selon les exemples, le temps d'analyse est plus grand avec l'une ou l'autre des méthodes.
3. L'option *min* permet d'éviter les appels inutiles de l'analyse dynamique de l'option *each*, mais classe moins d'alarmes, dans certains cas analysant des programmes de plus grande taille pouvant engendrer des dépassements du temps autorisé.
4. L'option *smart* classe autant d'alarmes que l'option *each* en évitant les appels inutiles de l'analyse dynamique, donc en améliorant le temps.

6.8 Évaluation statistique de l'apport du *slicing*

Les options *all*, *each*, *min* et *smart* utilisent le *slicing* pour simplifier le programme sous test. Dans cette partie, on étudie l'effet de cette simplification sur la longueur de chemins faisables, le nombre des chemins faisables et infaisables, la taille des programmes en nombre de lignes de code. Cette étude sera basée sur les mesures des critères $\mathcal{C}_5 - \mathcal{C}_7$. Ces critères nous permettent de quantifier la simplification des programmes analysés qui est apportée par le *slicing* à la fois sur le plan statique (nombre de lignes des programmes) et sur le plan dynamique (longueur des chemins, nombre de chemins à examiner). La longueur d'un chemin est le nombre des nœuds de branchement rencontrés tout le long du chemin. Dans cette partie, nous faisons également une étude des taux moyens de réduction des alarmes non classées et de la durée de l'analyse des programmes en utilisant les mesures des critères $\mathcal{C}_1 - \mathcal{C}_4$ faites précédemment.

N°	*all-threats DA*	SANTE *none*	SANTE *all*		SANTE *each*		SANTE *min*		SANTE *smart*	
			longueur	taux	longueur	taux	longueur	taux	longueur	taux
1	90	43	43	0%	43	0%	43	0%	43	0%
2	80	64	63	2%	49	24%	63	2%	63	2%
3	-	70	70	0%	53	25%	52	26%	52	26%
5	27	20	20	0%	20	0%	20	0%	20	0%
6	56	39	39	0%	39	0%	39	0%	39	0%
7	996	526	525	1%	153	71%	153	71%	153	71%
9	6	6	6	0%	6	0%	6	0%	6	0%

FIGURE 6.8 – Longueur moyenne des chemins des contre-exemples

Contre-exemples simplifiés

L'utilisation du *slicing* dans SANTE supprime le code non pertinent pour les alarmes analysées. Les contre-exemples trouvés exécutent des chemins beaucoup plus courts sur des programmes simplifiés, comme le montre la figure 6.8. Elle indique aussi le taux de réduction de chaque option, par rapport à l'option *none* qui n'utilise pas le *slicing*.

Le taux de réduction de longueur des chemins des contre-exemples est calculé par la formule suivante :

$$t = 100 - \left(\frac{longueur_{option} * 100}{longueur_{none}} \right)$$

où $longueur_{option}$ est la longueur des chemins des contre-exemples avec une option utilisant le *slicing* (*all*, *each*, *min* et *smart*) et $longueur_{none}$ est la longueur des chemins des contre-exemples avec l'option *none*, qui n'effectue pas de simplification syntaxique.

dans la figure 6.8, on ne présente la longueur moyenne des contre-exemples ni pour l'exemple 4 parce qu'il dépasse le temps limite avant de détecter aucun bogue avec l'option *none* ni pour l'exemple 8 parce qu'aucun bogue n'a été détecté dans cet exemple.

Dans nos expérimentations, la longueur du chemin des contre-exemples diminue en moyenne de 12%. Ce taux atteint 71% sur l'exemple 7, en particulier avec l'option *each* et avec les options avancées (*min* et *smart*).

Il convient de préciser que les exemples 1 et 5 sont des versions simplifiées du code réel afin de mieux illustrer les erreurs détectées. Cela explique le faible taux de réduction par le *slicing*. La moyenne du taux de réduction de longueur des chemins des contre-exemples sur les exemples non simplifiés 2, 3, 6, 7 et 9 serait de 16%.

N°	*all-threats DA*	SANTE *none*	SANTE *all*		SANTE *each*		SANTE *min*		SANTE *smart*	
			longueur	taux	longueur	taux	longueur	taux	longueur	taux
1	86	39	39	0%	37	6%	39	0%	39	0%
2	63	52	51	2%	32	39%	51	2%	51	2%
3	-	66	66	0%	49	26%	48	28%	48	28%
5	67	50	50	0%	50	0%	50	0%	50	0%
7	1092	560	560	0%	106	82%	106	82%	106	82%
9	129	77	61	21%	58	25%	59	24%	59	24%

FIGURE 6.9 – Longueur moyenne des chemins explorés

6.8.1 Chemins simplifiés

La figure 6.9 présente la longueur moyenne des chemins parcourus par les cas de tests exécutés avec chaque méthode ainsi que le taux de réduction de chaque option par rapport à l'option *none* qui n'utilise pas le *slicing*. Nous avons effectué cette évaluation uniquement pour les exemples 1, 2, 3, 5, 7 et 9 sur lesquels l'analyse termine normalement (sans *time / space out*).

La longueur des chemins exécutés diminue après l'utilisation du *slicing* avec un taux moyen de 19%. Ce taux atteint 82% sur l'exemple 7, en particulier avec l'option *each* et avec les options avancées (*min* et *smart*) (voir figure 6.9 exemple 7). Ce taux est calculé par la même formule que précédemment.

6.8.2 Réduction du nombre de chemins

Dans le pire des cas, le nombre de chemins (faisables et infaisables) est exponentiel relativement à la taille du programme exprimée en nombre de lignes de code. Ainsi, même une légère réduction du programme par le *slicing* avant la génération de tests peut diminuer considérablement le nombre de chemins.

Le taux de réduction de la taille des programmes est calculé par la formule suivante :

$$t = 100 - \left(\frac{taille_{slice} * 100}{taille_{none}} \right)$$

où $taille_{slice}$ est la taille (en nombre de lignes de code) de chaque *slice* avec une option utilisant le *slicing* et $taille_{none}$ est la taille (en nombre de lignes de code) du programme initial.

La figure 6.10 présente la taille des programmes en nombre de lignes de code et des *slices* analysées par PATHCRAWLER avec chaque méthode. Elle montre que le taux moyen

N°	*all-threats DA*	*none*	*all*	*each*	*min*	*smart*
1	33	33	32	28, 28, 32, 32,	32	32
2	97	97	94	45, 51, 54, 90, 94	94	94
3	55	55	55	32, 32, 32, 37, 37, 38, 38, 38 38, 39, 39, 39 39, 39, 39	37, 38, 39, 39	37, 38, 39, 39
4	696	168	154	73, 73, 84, 84, 114, 114, 114, 114, 114, 114, 114, 114	154	154, 84
5	50	50	50	50, 50, 50, 50	50	50
6	705	181	151	131, 131, 131, 131, 131, 131,131, 132, 132, 133, 140, 140	132, 132, 133, 140	132, 132, 133, 140,
7	202	179	96	20, 20, 29, 29, 30, 30, 31, 33, 34, 34	20, 20, 29, 29, 30, 30, 31, 33, 34, 34	20, 20, 29, 29, 30, 30, 31, 33, 34, 34
8	365	83	81	22, 81	81	81, 22
9	154	124	74	20, 20, 32, 60, 62	20, 62	20, 62

FIGURE 6.10 – Réduction de la taille du code en nombre de lignes

de réduction de la taille des programmes avec l'option *all* est d'environ 14% et qu'il atteint 47% sur l'exemple 7. Le taux moyen devient 26% avec l'option *min*, 32% avec l'option *smart* et 37% avec l'option *each*. Ce taux atteint 89% pour certaines alarmes dans l'exemple 7.

6.8.3 Classification des alarmes

Par rapport à *all-threats DA*, le taux de réduction du nombre d'alarmes non classées est calculé par la formule suivante :

$$t = 100 - \left(\frac{U_{option} * 100}{U_{all-threats\ DA}} \right)$$

où U_{option} est le nombre d'alarmes restantes non classées avec SANTE en utilisant une des options fournies (*none*, *all*, *each*, *min* et *smart*) et $U_{all-threats\ DA}$ est le nombre d'alarmes restantes non classées avec *all-threats DA*.

Ce taux est de 53% pour l'option *none*. Il devient 59% avec l'option *all*, 80% avec l'option *min* et 89% avec les options *each* et *smart*.

N°	*all-threats DA*	*none*	*all*	*each*	*min*	*smart*
1	3811	1515	1515	941, 941, 1515, 1515	1515	1515
2	3876	2141	2141	35, 98, 268, 1696, 2141	2141	2141
5	17951	14223	14223	14223, 14223, 14223, 14223	14223	14223
7	23428	12530	12530	1, 1, 47, 47, 47, 47, 289, 313, 313, 313	1, 1, 47, 47, 47, 47, 289, 313, 313, 313	1, 1, 47, 47, 47, 47, 289, 313, 313, 313
9	732	428	286	3, 3, 12, 218, 282	3, 218	3, 218

FIGURE 6.11 – Nombre des chemins infaisables

Par rapport à l'analyse de valeurs, le taux de réduction est calculé par la formule suivante :

$$t = 100 - \left(\frac{U_{option} * 100}{U_{VA}}\right)$$

où U_{option} est le nombre d'alarmes restantes non classées avec SANTE en utilisant une des options fournies et U_{VA} est le nombre d'alarmes restantes non classées avec l'analyse de valeurs seule.

Ce taux est de 64% pour l'option *none*. Il devient 69% avec l'option *all*, 85% avec l'option *min* et 92% avec les options *each* et *smart*. Pour certains exemples, le taux de réduction du nombre d'alarmes non classées pourrait atteindre 100%, lorsque toutes les alarmes seraient classées. Ainsi, le nombre d'alarmes restantes non classées est plus petit avec l'option *smart* et avec l'option *each*, mais l'option *each* consomme plus de temps.

6.8.4 Durée d'analyse

SANTE consomme moins de temps que PATHCRAWLER *all-threats*, et il permet d'éviter certains *time / space out*. Le processus de vérification devient particulièrement rapide avec les options avancées (*min* et *smart*).

Le taux de réduction de temps d'analyse est calculé par la formule suivante :

$$t = 100 - \left(\frac{temps_{option} * 100}{temps_{all-threatsDA}}\right)$$

où $temps_{option}$ est le temps consommé par SANTE avec une des options fournies (*none*, *all*, *each*, *min* et *smart*) et $temps_{all-threatsDA}$ est le temps consommé par *all-threats DA*.

Le temps consommé diminue d'environ 43%, et cette diminution peut atteindre 98% sur certains exemples avec les options avancées.

Ce gain de temps est dû à :

1. la réduction du nombre de chemins faisables dans le programme sous test,
2. la réduction de la taille des chemins et
3. la réduction du nombre de chemins infaisables, comme le montre la figure 6.11, qui présente le nombre de chemins infaisables pour chaque méthode et chaque programme analysé, uniquement, pour les exemples 7 et 9 sur lesquels l'analyse termine normalement (sans *time / space out*).

6.8.5 Bilan de cette expérimentation

Les principales conclusions de cette section sont :

1. L'utilisation du *slicing* permet de signaler les erreurs avec des informations plus précises, les illustrer sur un programme plus simple avec un chemin d'erreur plus court qui montre uniquement les contraintes sur les valeurs des variables utilisées pour le calcul de l'instruction menaçante. Dans nos expérimentations, les options utilisant le *slicing* permettent de diminuer la longueur des chemins des contre-exemples en moyenne de 12%, ce taux allant jusqu'à 71% sur certains exemples. La longueur des chemins traités lors de l'analyse dynamique des programmes simplifiés est réduit en moyenne de 19%, ce taux pouvant atteindre 82% sur un de nos exemples. La durée d'analyse diminue également de 43% en moyenne.

2. Le taux de réduction d'alarmes non classées avec SANTE par rapport à *all-threats DA* varient entre 53% (pour l'option *none*) jusqu'à 89% (pour les options *each* et *smart*).

6.9 Conclusion - Les hypothèses sont-elles satisfaites ?

Dans ce chapitre nous avons présenté des expérimentations pour valider la méthode proposée. Les principaux résultats de ce chapitre sont :

1. L'hypothèse $\mathcal{H}_1$ est confirmée par nos expérimentations. Le bilan dans la partie 6.6.5 montre que l'outil SANTE combinant l'analyse de valeurs et la génération de tests structurels est plus performant que chacun de ces outils utilisés séparément. Il classe plus d'alarmes qu'un analyseur statique et il est plus efficace en temps qu'un outil de test structurel.

2. L'hypothèse $\mathcal{H}_2$ est également confirmée par nos expérimentations. Comme nous l'avons vu dans les parties 6.7.3 et 6.8.5, le *slicing* permet de supprimer automatiquement du code non pertinent et permet d'accélérer la phase de génération de

tests. Il peut donc éliminer certains *time / space out* à la génération de tests et permettre de classer plus d'alarmes.

3. L'hypothèse $\mathcal{H}_3$ est confirmée si on l'évalue sur l'ensemble des exemples, mais elle ne l'est pas au cas par cas. La partie 6.7.3 montre que sur l'ensemble des exemples, l'option *smart* classe autant d'alarmes que l'option *each*. Globalement elle met moins de temps. Mais il est possible sur un exemple que l'option *smart* ne classe pas plus d'alarmes que l'option *min* ou *all* et fasse plus de dépassements de temps limite. Dans ce cas, la méthode *smart* peut s'avérer moins performante que les méthodes *all* ou *min*. Pour ce type d'exemple, l'hypothèse $\mathcal{H}_3$ serait invalidée. Cependant, comme nous l'avons vu dans le Chapitre 4 l'option *smart* ne perd pas de temps inutilement : elle continue les itérations tant qu'il reste des alarmes pouvant théoriquement être classées, et elle ne fait jamais de travail superflu. En plus, l'option *smart* nous garantit pour toute alarme qu'elle n'a pas pu classer (dans un délai fixé) qu'il n'est pas possible de la classer avec les autres options (dans le même délai fixé).

4. L'hypothèse $\mathcal{H}_4$ est confirmée par nos expérimentations. Comme nous l'avons vu dans la partie 6.8.5, le *slicing* permet d'illustrer l'erreur détectée sur un programme simplifié, donc avec un chemin d'exécution plus court et débarassé d'informations qui n'ont pas de rapport avec l'erreur.

5. L'amélioration du taux d'alarmes classées avec SANTE est très clair. Cependant, il y a des alarmes qui restent non classées (exemples 4 et 8). De plus, même avec les options *each* et *smart*, qui donnent les meilleurs résultats en terme de classification des alarmes, il y a une perte de temps et SANTE *smart* peut faire plusieurs *time / space out* avant de conclure. L'option *smart* réduit les ensembles d'alarmes traités après chaque itération en enlevant les alarmes finales à chaque fois. Une sorte de réduction "dichotomique" des ensembles d'alarmes pourrait réduire ce nombre de *time / space out* en divisant ces ensembles au lieu d'enlever les alarmes finales seulement. Nous développons ce point en perspectives.

Chapitre 7

Conclusions et perspectives

Sommaire

Dans ce chapitre, nous concluons la thèse en rappelant les objectifs principaux du travail, en décrivant les résultats obtenus et en proposant des perspectives d'extension.

7.1 Rappel des objectifs

Notre objectif principal était de fournir une méthode de vérification de programmes écrits en langage C, automatique ou ne nécessitant que peu d'expertise de l'utilisateur. Un autre objectif était de fournir à l'ingénieur validation des informations précises sur l'erreur détectée, permettant de réduire le temps d'analyse et de correction de l'erreur. Après avoir identifié les forces et les faiblesses de l'analyse statique et de l'analyse dynamique, il est clairement apparu que notre objectif passait par la conception de combinaisons originales de ces deux sortes d'analyses, ainsi que de la simplification de programmes pour garder uniquement les informations pertinentes sur l'erreur détectée.

L'intégration de la simplification de programmes a également permis de répondre au deuxième objectif. En effet, la simplification de programmes par rapport aux alarmes fournit une version simplifiée du programme initial contenant les informations les plus précises

nécessaires pour analyser les alarmes et les erreurs détectées. L'origine d'une éventuelle erreur est plus facile à comprendre sur un programme plus simple après avoir enlevé les parties du code qui ne sont pas pertinentes.

7.2 Bilan

Nous avons présenté une méthode originale combinant une analyse de valeurs, une analyse de dépendances, une simplification syntaxique et du test structurel. Cette méthode combine la sûreté de l'interprétation abstraite avec la précision de l'analyse dynamique.

Dans une première combinaison sans simplification syntaxique, l'analyse statique signale les instructions risquant de provoquer des erreurs à l'exécution, par des alarmes dont certaines peuvent être de fausses alarmes, puis l'analyse dynamique est utilisée pour confirmer ou rejeter ces alarmes. Cette combinaison souffrait du manque de ressources avec les programmes de grande taille.

Face à ce problème, nous avons intégré la simplification syntaxique dans notre combinaison. La simplification syntaxique est utilisée entre l'analyse statique et l'analyse dynamique pour simplifier le programme initial par rapport à ses alarmes. Quatre utilisations de la simplification syntaxique ont été étudiées. Les deux premières sont présentées comme les utilisations de base *all* et *each*. L'utilisation *all* simplifie le programme en tenant compte de toutes les alarmes trouvées par l'analyse de valeurs. L'utilisation *each* simplifie le programme en tenant compte de chaque alarme séparément. Ensuite nous avons étudié les propriétés des dépendances entre alarmes et nous avons proposé deux autres utilisations, optimisées et adaptatives, appelées *min* et *smart*. Dans l'utilisation *min*, le programme est simplifié par rapport à un ensemble minimal de sous-ensembles d'alarmes. L'union de ces sous-ensembles couvre l'ensemble de toutes les alarmes. L'utilisation *smart* applique l'utilisation précédente itérativement en réduisant progressivement la taille des sous-ensembles.

La méthode fournit un contre-exemple pour chaque erreur détectée. Elle illustre cette erreur par un programme plus simple, avec un chemin plus court et avec un ensemble réduit de contraintes conduisant à l'instruction erronée, portant sur les variables utiles seulement. Nous avons présenté cette méthode de façon détaillée et nous avons prouvé sa correction.

Nous avons implémenté les travaux dans la plate-forme Frama-C. Cette implémentation utilise les greffons d'analyse de valeurs, d'analyse de dépendances et de *slicing* de Frama-C, puis l'outil de génération de tests structurels PathCrawler. Elle prend en entrée le programme à analyser et une précondition. Elle produit une liste d'alarmes et une classification safe ou bug pour chaque alarme. Certaines alarmes peuvent rester non classées (unknown).

Nous avons effectué des expérimentations sur des programmes existants. Ces expérimentations ont illustré les apports de la méthode mais ont également montré ses limites. Les apports se résument par l'augmentation du taux de classification des alarmes, la réduction du temps d'analyse et la remise d'informations plus précises sur l'erreur détectée. Les limites sont mises en évidence par le taux des alarmes restantes non classées et la perte de temps qui peut arriver dans certains cas.

7.2.1 Publications associées à la thèse

Dans le cadre de nos travaux, nous avons réalisé six communications de recherche.

1. [CKGJ10a] – Workshop français – GDR GPL 2010
2. [Che10] – Conférence française – AFADL 2010
3. [CKGJ10b] – Conférence internationale – TAP 2010
4. [CKGJ11] – Conférence internationale – TAP 2011
5. [CKGJ12] – Conférence internationale – SAC 2012
6. [KBR+12] – Conférence internationale – TACAS 2012

L'article [CKGJ10a] présente la première combinaison d'analyse de valeurs et d'analyse dynamique sans simplification syntaxique. Cet article a été étendu pour faire l'objet d'une publication dans une conférence internationale [CKGJ10b] après l'ajout des expérimentations qui ont permis d'évaluer la méthode.

L'article [Che10] présente l'outil SANTE CONTROLLER implémenté dans la plate-forme FRAMA-C. Cette implémentation utilise les greffons d'analyse de valeurs et de *slicing* de FRAMA-C et l'outil de génération de tests structurels PATHCRAWLER.

L'article [CKGJ11] présente l'intégration de la simplification syntaxique dans la méthode combinant l'analyse de valeurs et l'analyse dynamique. Il présente également les deux utilisations *all* et *each* basiques de la simplification syntaxique.

L'article [CKGJ12] présente les deux utilisations avancées de la simplification syntaxique exploitant les dépendances entre menaces. Il présente également une formalisation de la méthode, ainsi que l'évaluation de chaque utilisation.

L'article [KBR+12] présente la nouvelle version de PATHCRAWLER qui intègre le greffon *PC Analyzer*.

7.3 Perspectives

Les diverses techniques de vérification étudiées et les expérimentations effectuées au cours de cette thèse suggèrent quelques améliorations et ouvrent de nouvelles perspectives

pour des travaux futurs.

7.3.1 Prendre en compte d'autres types de menaces

Une amélioration immédiate serait d'étendre la méthode et l'implémentation pour traiter d'autres types d'erreurs possibles, comme la lecture d'une variable non initialisée, le débordemment arithmétique et le traitement des cas manquants d'erreurs de pointeurs.

Ces améliorations permettraient de tester une plus grande classe de programmes. Nous pourrions valider la méthode sur d'autres exemples industriels.

7.3.2 Combiner la preuve et l'analyse dynamique

Combiner la preuve avec l'analyse dynamique est une autre perspective. Nous pourrions utiliser les greffons WP et / ou Jessie de Frama-C. En cas d'échec de la preuve, nous pourrions étudier une analyse des raisons de l'échec et transmettre ces raisons à l'analyse dynamique. Celle-ci chercherait alors un cas de test confirmant que l'échec est dû à un réel cas d'erreur. Cette voie paraît intéressante mais soulève des problèmes difficiles qui sont la formalisation des raisons de l'échec d'une preuve, l'extraction de cette information des traces de preuve et leur utilisation par le générateur de tests. Une thèse est proposée pour étudier cette problématique.

7.3.3 Améliorer la simplification syntaxique

Nous avons proposé dans ce mémoire quatre utilisations de la simplification syntaxique, dont la plus puissante est l'utilisation *smart*, qui réduit les sous-ensembles d'alarmes après chaque itération en enlevant les alarmes finales. Il serait intéressant de la comparer avec une sorte de réduction "dichotomique" des sous-ensembles d'alarmes en les divisant suivant le niveau de dépendances d'alarmes au lieu d'enlever juste les alarmes finales.

7.3.4 Améliorer l'analyse statique

Dans l'état actuel, sante utilise l'analyse de valeurs avec une configuration qui consiste à calculer un seul état de tous les cas possibles à l'exécution à chaque point de programme. Cette configuration est celle qui consomme le moins de temps, mais elle est aussi la moins précise. Une configuration qui consiste à calculer plusieurs états à chaque point de programme pourrait peut-être éliminer certaines alarmes. Une piste de travail intéressante serait d'expérimenter la méthode sante avec d'autres configurations de l'analyse de valeurs afin de trouver un compromis raisonnable et acceptable entre précision de la classification des alarmes et temps d'analyse du programme.

7.3.5 Améliorer l'analyse dynamique

Dans nos expériences, nous avons comparé les méthodes pour une profondeur k du critère k-chemins qui est la valeur minimale qui permet de détecter toutes les erreurs connues sur les exemples choisis avec SANTE. L'exemple 4 illustre un cas où le critère "tous les k-chemins" permet de trouver trois erreurs plus qu'avec le critère "tous les chemins" avec SANTE *none*. C'est en effet de la limitation de la recherche en profondeur. La détermination de k reste donc un problème à résoudre pour deux raisons. Premièrement nous cherchons cette valeur séquentiellement, ce qui n'est pas efficace. Plus grave, dans une application réelle, ne connaissant pas le nombre d'erreurs connues, nous ne pourrons plus utiliser cette information comme critère d'arrêt. Donc la définition d'une méthode pour choisir k reste un travail à faire avec pour objectif de trouver un compromis raisonnable et acceptable entre précision de la classification des alarmes et temps d'analyse du programme.

Actuellement, l'analyse dynamique effectue un parcours de "tous les chemins" ou de "tous les k-chemins" en profondeur d'abord. Il faudrait davantage d'expérimentations avec d'autres types de parcours, comme le parcours générationnel de tous les chemins utilisé dans SAGE [GLM08]. Ce parcours est plus adapté à la détection des erreurs, parce qu'il permet d'exploiter toutes les branches en même temps. Il serait aussi intéressant d'expérimenter avec un parcours de toutes les branches.

Annexes

Annexe A

Extrait de code source de SANTE CONTROLLER

La figure A.1 présente sommairement la fonction principale de SANTE CONTROLLER. La conditionnelle à la ligne 2 vérifie si l'option `-sante` est sélectionnée. Les lignes 4 et 36 marquent le début et la fin du traitement. Les lignes 5–6 créent un nouveau projet nommé `sante` et copient l'arbre de syntaxe abstraite normalisé dans ce projet. À la ligne 8, on prend le nouveau projet créé comme projet courant. La ligne 9 appelle l'analyse de valeurs. Les lignes 10–11 collectent les alarmes signalées et les stockent dans une liste. À la ligne 12, on vérifie si l'option `-slice-all` est renseignée. Dans ce cas on exécute le traitement de l'option *all* (lignes 13–16) qui consiste à appeler la fonction `runAll` (ligne 14) et le calcul de diagnostics par la fonction `getDiagnostics` (ligne 15). Pour l'option *each* (lignes 17–21), on appelle la fonction `runEach` (ligne 19). Le calcul de diagnostics est effectué par la fonction `getDiagnosticsForEach` (ligne 20) qui fusionne les diagnostics trouvés par les différentes sessions d'analyse dynamique. La même chose pour les options *min* (lignes 22–25) et *smart* (lignes 26–29) où la fusion des diagnostics est effectué dans les fonctions `runMin` (ligne 24), respectivement `runSmart` (ligne 28). Les lignes 30–35 correspondent à l'option *none*, qui consiste à appeler *PC Analyzer* (ligne 32) et ensuite lancer la génération de tests (ligne 33) directement sans passer par le *slicing*. La ligne 34 calcule les diagnostics.

La figure A.2 présente sommairement la fonction `runAll`. Elle procède ainsi :

1. elle crée un nouveau projet nommé `sliceAll` (lignes 2 – 3),
2. elle exécute un *slicing* par rapport à la liste `alarms` des alarmes dans le nouveau projet (ligne 4),
3. elle appelle *PC Analyzer* (ligne 5) et
4. elle lance la génération de tests (ligne 6).

La figure A.3 présente sommairement la fonction récursive `runEach`. Elle prend une liste d'alarmes en entrée ainsi que le projet principal et un compteur. Elle procède ainsi :

1. elle initialise le critère de *slicing* `criterion`. Avec cette option, le crière du *slicing* est une alarme différente à chaque fois parmi la liste passée en paramètre. (ligne 5),

```
1  let startup _ =
2    if Options.Enabled.get () then
3      begin
4        Format.printf "[SANTE] started...@.";
5        let proj = File.create_project_from_visitor "sante"
6           (fun proj -> new Visitor.frama_c_copy proj)
7        in
8          Project.set_current proj;
9          !Db.Value.compute ();
10         let alarms = ref [] in
11           getAlarms alarms;
12           if SliceAll.get() then
13             begin
14               runAll !alarms
15               getDiagnostics alarms;
16             end
17           else if SliceEach.get() then
18             begin
19               runEach !alarms proj 1
20               getDiagnosticsForEach alarms;
21             end
22           else if SliceMin.get() then
23             begin
24               runMin !alarms proj
25             end
26           else if SliceSmart.get() then
27             begin
28               runSmart !alarms proj
29             end
30           else
31             begin
32               !Db.pcanalyzer.compute ();
33               runTestGeneration ();
34               getDiagnostics alarms;
35             end;
36           Format.printf "[SANTE] finished...@.";
37   end
```

FIGURE A.1 – Fonction principale de SANTE CONTROLLER

```
1 let runAll alarms =
2   let newProject = "sliceAll" in
3     !Db.Slicing.Project.mk_project newProject
4     slice !alarms newProject;
5     !Db.pcanalyzer.compute ();
6     runTestGeneration ();
```

FIGURE A.2 – Fonction `runAll` qui implémente l'option *all*

```
1 let rec runEach alarms mainProject nb =
2   match alarms with
3     | [] -> ()
4     | head :: tail ->
5         let criterion = ref [head] in
6           Project.set_current mainProject;
7           let newProject = "slice"^(string_of_int nb) in
8             !Db.Slicing.Project.mk_project newProject
9             slice !criterion newProject;
10            !Db.pcanalyzer.compute ();
11            runTestGeneration ();
12            runEach tail mainProject (nb+1)
```

FIGURE A.3 – Fonction `runEach` qui implémente l'option *each*

2. elle prend le projet principal `mainProject` comme projet principal (ligne 6),
3. elle crée un nouveau projet (lignes 7 et 8) pour y mettre le programme simplifié,
4. elle effectue le *slicing* (ligne 9) du programme initial par rapport au critère `criterion` et sauvegarde le programme résultant dans le nouveau projet crée dans l'action précédente,
5. elle appelle *PC Analyzer* (ligne 10),
6. elle lance la génération de tests sur le programme simplifié (ligne 11) et
7. elle appelle la fonction `runEach` récursivement en lui donnant en entrée la liste des alarmes restantes, le projet principal et le compteur incrémenté (ligne 12). Elle termine quand toutes les alarmes sont traitées et la liste des alarmes restantes devient vide (ligne 3).

L'implémentation des fonctions `runMin` et `runSmart` n'est pas détaillée ici, mais elle suit respectivement les algorithmes 1 et 2 présentés dans le chapitre 4.

Bibliographie

[AC03] Jean-Raymond Abrial and Dominique Cansell. Click'n prove : Interactive proofs within set theory. In TPHOLs, pages 1–24, 2003.

[ALN+91] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The B-Method. In VDM Europe (2), pages 398–405, 1991.

[Ame89] American National Standards Institute. American National Standard Programming Language C, ANSI X3.159-1989, 1989.

[AST11] The Astrée Static Analyzer, 2011. http ://www.astree.ens.fr/.

[Bal04] Thomas Ball. A theory of predicate-complete test coverage and generation. In FMCO, pages 1–22, 2004.

[BBD+10] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. Theor. Comp. Sci., 411(11–13) :1372–1386, 2010.

[BC04] Yves Bertot and Pierre Casteran. Interactive Theorem Proving and Program Development. SpringerVerlag, 2004.

[BCDL05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, and Bart Jacobsand K. Rustan M. Leino. Boogie : A modular reusable verifier for object-oriented programs. In FMCO, pages 364–387, 2005.

[BDHTH+09] Bernard Botella, Mickaël Delahaye, Stéphane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs : Experience with PathCrawler. In AST, pages 70–78, Vancouver, Canada, May 2009. IEEE Computer Society.

[Bei90] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, New-York, 1990.

[Ber08] Yves Bertot. A short presentation of coq. In TPHOLs, pages 12–16, 2008.

[BFH+08] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL : ANSI C Specification Language (preliminary design V1.2), preliminary edition, May 2008.

[BGP+09] Olivier Bouissou, Eric Goubault, Sylvie Putot, Karim Tekkal, and Franck Védrine. HybridFluctuat : A static analyzer of numerical programs within

a continuous environment. In CAV, volume 5643 of LNCS, pages 620–626, 2009.

[BH08] Sébastien Bardin and Philippe Herrmann. Structural testing of executables. In ICST'08, pages 22–31, Lillehammer, Norway, April 2008.

[BHJM05] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In FASE, volume 3442 of LNCS, pages 2–18. Springer, 2005.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast : Applications to software engineering. Int. J. Softw. Tools Technol. Transfer, 9(5-6) :505–525, 2007.

[BLA11] BLAST : Berkeley Lazy Abstraction Software Verification Tool, 2011. http ://mtc.epfl.ch/software-tools/blast/index-epfl.php.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In IN PROC. ACM PLDI, pages 203–213. ACM Press, 2001.

[BNR+10] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. Proofs from tests. IEEE Trans. Software Eng., 36(4) :495–508, 2010.

[BOO11] Boogie : An Intermediate Verification Language, 2011. http ://research.microsoft.com/en-us/projects/boogie/.

[Bou08] Olivier Bouissou. Analyse statique par interprétation abstraite de systèmes hybrides. These, Ecole Polytechnique X, September 2008.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project : debugging system software via static analysis. In POPL, pages 1–3, 2002.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL, pages 238–252, New York, NY, USA, 1977. ACM.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. J. Log. Comput., 2(4) :511–547, 1992.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In ESOP, volume 3444 of LNCS, pages 21–30. Springer, April 2005.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, and Xavier Rival. Why does Astrée scale up ? FMSD, 35(3) :229–264, December 2009.

[CCG03] Sagar Chaki, Edmund Clarke, and Alex Groce. Modular verification of software components in C. In IEEE Transactions on Software Engineering, pages 385–395, 2003.

[CCKL07] Sylvain Conchon, Evelyne Contejean, Johannes Kannig, and Stéphane Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In John Rushby and N. Shankar, editors, AFM, pages 55–59, New York, NY, USA, 2007. ACM.

[CCM09] Geraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. IEEE International Workshop on Source Code Analysis and Manipulation, pages 123–124, 2009.

[CDA11] Loic Correnson, Zaynah Dargaye, and Pacalet Anne. WP : Plug-in Manual, preliminary edition, February 2011. http ://frama-c.com/download/frama-c-wp-manual.pdf.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE : Unassisted and automatic generation of high-coverage tests for complex systems programs, 2008.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of Programs, Workshop, pages 52–71, London, UK, 1982. Springer-Verlag.

[CG10] Florence Charreteur and Arnaud Gotlieb. Constraint-Based Test Input Generation for Java Bytecode. In ISSRE, San Jose, CA, USA, 2010.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In CAV, volume 1855 of LNCS, pages 154–169. Springer, 2000.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE : Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur., 12(2), 2008.

[Cha10] Florence Charreteur. Modélisation par contraintes de programmes en bytecode Java pour These, Université Européenne de Bretagne, March 2010.

[Che10] Omar Chebaro. Outil SANTE : Détection d'erreurs par analyse statique et test structurel des programmes C. In AFADL, Poitiers, France, 2010.

[CIL06] CIL : C Intermediate Language. Infrastructure for C Program Analysis and Transformation. CIL, 2005/2006. http ://manju.cs.berkeley.edu/cil/.

[CK04] David R. Cok and Joseph R. Kiniry. ESC/Java2 : Uniting ESC/Java and JML : Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In CASSIS. Springer–Verlag, 2004.

[CKGJ10a] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Combining Frama-C and PathCrawler for C program debugging. extended abstract. In GDR GPL, pages 217–218, Pau, France, 2010.

[CKGJ10b] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Combining static analysis and test generation for C program debugging. In TAP, volume 6143 of LNCS, pages 652–666. Springer, 2010.

[CKGJ11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool : Value analysis, program slicing and test generation for C program debugging. In TAP, pages 78–83, 2011.

[CKGJ12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In SAC, Trento, Italy, 2012. To appear.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 2(3) :215–222, 1976.

[Cod11] Happy Codings. Program to calculate Area of a Polygon, 2010–2011. http ://www.c.happycodings.com/Mathematics/code4.html.

[COQ11] The Coq Proof Assistant, 2011. http ://coq.inria.fr/.

[CPR06a] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. SIGPLAN Not., 41 :415–426, June 2006.

[CPR06b] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator : Beyond safety. In CAV, volume 4144 of LNCS, pages 415–418. Springer, 2006.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher : An automatic robustness tester for Java. Software Practice and Experience, 34(11) :1025–1050, September 2004.

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash : Combining static checking and testing. In ICSE, pages 422–431. ACM, May 2005.

[CS06] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher : a hybrid analysis tool for bug finding. In ISSTA, pages 245–254. ACM, 2006.

[CS09] Pascal Cuoq and Julien Signoles. Experience report : Ocaml for an industrial-strength static analysis framework. In ICFP, pages 281–286, 2009.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM, 18 :453–457, August 1975.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In TACAS, volume 4963 of LNCS, pages 337–340. Springer, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. J. ACM, 52 :365–473, May 2005.

[DOS99] Chris DiBona, Sam Ockman, and Mark Stone, editors. Open Sources : Voices from the Open Source Revolution. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27 :99–123, 2001.

[Ecl11] The Eclipse Foundation. Eclipse, 2011. http ://www.eclipse.org/.

[Ent07] Nicholas Enticknap. It salary survey : skills gap pushes up pay for key roles, 11 2007. Computer Weekly.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program., 69(1–3) :35–45, 2007.

[ERG11] The Alt-Ergo theorem prover, 2011. http ://ergo.lri.fr/.

[Ern03] Michael D. Ernst. Static and dynamic analysis : Synergy and duality. In WODA 2003 : ICSE Workshop on Dynamic Analysis, pages 24–27, Portland, OR, May 9, 2003.

[Fal01] Rainer Faller. Project experience with iec 61508 and its consequences. In SAFECOMP, volume 2187 of LNCS, pages 200–214. Springer, 2001.

[Fer04] Jérôme Feret. Static analysis of digital filters. In ESOP, number 2986 in LNCS. Springer-Verlag, 2004.

[Flo63] Robert W. Floyd. Syntactic analysis and operator precedence. J. ACM, 10(3) :316–333, 1963.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In ICFEM, volume 3308 of LNCS, pages 15–29. Springer, 2004.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In CAV, volume 4590 of LNCS, pages 173–177. Springer, 2007.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9 :319–349, 1987.

[Fra11] Frama-C. Framework for static analysis of C programs, 2007–2011. http ://frama-c.com/.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In CAV'07, pages 322–335, Berlin, Germany, July 2007.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY : a new algorithm for property checking. In SIGSOFT FSE, pages 117–127. ACM, 2006.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed automated random testing. In PLDI'05, pages 213–223, Chicago, IL, USA, June 2005.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In NDSS. The Internet Society, 2008.

[GM93] M. J. C. Gordon and T. F. Melham, editors. Introduction to HOL : A theorem proving environment for higher order logic. Cambridge University Press, 1993.

[God07] P. Godefroid. Compositional dynamic test generation. SIGPLAN Not., 42(1) :47–54, 2007.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In SAS, volume 4134 of LNCS, pages 18–34. Springer, 2006.

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In VMCAI, volume 6538 of LNCS, pages 232–247. Springer, 2011.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In CAV, volume 1254 of LNCS, pages 72–83. Springer, 1997.

[Guo06] Philip J. Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. In Master's thesis, MIT Dept. of EECS, 2006.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In Proceedings of the IEEE, pages 1305–1320, 1991.

[Her11] Philippe Hermann. Frama-C's : annotation generator plug-in, preliminary edition, December 2011. http ://frama-c.com/download/frama-c-rte-manual.pdf.

[HJM+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In CAV, pages 526–538, London, UK, 2002. Springer-Verlag.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10) :567–580, 1969.

[HOL11] HOL4, 2011. http ://hol.sourceforge.net/.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12 :26–60, 1990.

[IA85] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1985.

[Ins90] The Institute of Electrical and Eletronics Engineers. IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990, September 1990.

[ISA11] Isabelle, 2011. http ://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[ISO90] ISO. ISO/IEC 9899 :1990 : Programming languages — C. International Organization for Standardization, 1990.

[ISO99] ISO. ISO/IEC 9899 :1999 : Programming Languages — C. International Organization for Standardization, December 1999.

[KBR+12] Nikolai Kosmatov, Bernard Botella, Muriel Roger, Nicky Williams, Omar Chebaro, and Nicolas Dugué. Pathcrawler-online.com : a web service for c program testing. In TACAS, 2012. submitted.

[KHCL07] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In ASE, pages 389–392, USA, 2007. ACM.

[Kin76] James C. King. Symbolic execution and program testing. Commun. ACM, 19(7) :385–394, 1976.

[Kin11] KindSoftware. ESC/Java2, 2009–2011. http ://secure.ucd.ie/products/opensource/ESCJava2/.

[KL88] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. Inf. Process. Lett., 29(3) :155–163, 1988.

[Kos10] Nikolai Kosmatov. Artificial Intelligence Applications for Improved Software Engineering Development : New Prospects, chapter XI : Constraint-Based Techniques for Software Testing. IGI Global, 2010.

[KR78] B. W. Kernighan and D. M. Ritchie. The C programming language. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[Lan92] William Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS), 1(4) :323–337, 1992.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : a Java Modeling Language. In Formal Underpinnings of Java Workshop (at OOPSLA'98, 1998.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML : A Behavioral Interface Specification Language for Java. Technical report, Department of Computer Science, Iowa State University, 1999.

[LIS11] CEA LIST. Page d'accueil de l'outil Fluctuat, 2005–2011. http ://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html.

[LLF+96] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 flight 501 failure. Report, Ariane 501 Inquiry Board, Paris, July 1996.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench : a tool for evaluating and synthesizing multimedia and communications systems. In MICRO, pages 330–335. IEEE Computer Society, 1997.

[MA00] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In ASE'00, pages 229–237, Grenoble, France, September 2000.

[MM10] Claude Marché and Yannick Moy. Jessie : Plug-in Tutorial, preliminary edition, December 2010. http ://frama-c.com/download/jessie-tutorial-Carbon-20101202-beta2.pdf.

[Moy09] Y. Moy. Preuve automatique et modulaire de la sûreté de fonctionnement des programmes C. PhD thesis, Université de Paris-Sud, janvier 2009.

[Mye79] Glenford J. Myers. The Art of Software Testing. John Wiley and Sons, 1979.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured : type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst., 27 :477–526, May 2005.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer, 2002.

[NRTT09] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The Yogi Project : Software property checking via static analysis and testing. In TACAS, pages 178–181, Berlin, Heidelberg, 2009. Springer-Verlag.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. SIGPLAN Not., 19 :177–184, April 1984.

[Pol11] Polyspace. Polyspace embedded software verification, 1994–2011. http ://mathworks.com/products/polyspace/.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Symposium on Programming, pages 337–351, 1982.

[Ray98] Eric S. Raymond. Goodbye, "free software" ; hello, "Open-Source", February 1998.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc., 74 :358–366, 1953.

[Rut11] Barry Rutten. eurocheck, 2011. http ://freshmeat.net/projects/eurocheck.

[RY88] Thomas W. Reps and Wuu Yang. The semantics of program slicing. Technical report, University of Wisconsin, 1988.

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In CAV, volume 4144 of LNCS, pages 419–423. Springer, 2006.

[SHR99] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermal. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In in Proceedings of the 21st International Conference on Software Engineering, pages 432–441. ACM Press, 1999.

[SLA11] SLAM, 2011. http ://research.microsoft.com/en-us/projects/slam/.

[Sli11] The program slicing plugin of Frama-C, 2007–2011. http ://frama-c.com/slicing.html.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE : a concolic unit testing engine for C. In ESEC/FSE, pages 263–272. ACM, September 2005.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. White box test generation for .NET. In TAP'08, volume 4966 of LNCS, pages 133–153. Springer, April 2008.

[TS06] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded : Parameterized unit testing with symbolic execution. IEEE Software, 23(4) :38–47, 2006.

[WB89] R. B. Whitner and O. Balci. Guidelines for selecting and using simulation model verification techniques. In WSC, pages 559–568, New York, NY, USA, 1989. ACM.

[Wei81] Mark Weiser. Program slicing. In ICSE, pages 439–449. IEEE Computer Society, 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. Commun. ACM, 25(7) :446–452, 1982.

[WMM03] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of structural tests for C functions. In ICSSEA'03, Paris, October 2003.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler : automatic generation of path tests by combining static and dynamic analysis. In EDCC, volume 3463 of LNCS, pages 281–292. Springer, 2005.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In SIGSOFT FSE, pages 97–106. ACM, 2004.

Résumé

La validation des logiciels est une partie cruciale dans le cycle de leur développement. Deux techniques de vérification et de validation se sont démarquées au cours de ces dernières années : l'analyse statique et l'analyse dynamique. Les points forts et faibles des deux techniques sont complémentaires.

Nous présentons dans cette thèse une combinaison originale de ces deux techniques. Dans cette combinaison, l'analyse statique signale les instructions risquant de provoquer des erreurs à l'exécution, par des alarmes dont certaines peuvent être de fausses alarmes, puis l'analyse dynamique (génération de tests) est utilisée pour confirmer ou rejeter ces alarmes. L'objectif de cette thèse est de rendre la recherche d'erreurs automatique, plus précise, et plus efficace en temps.

Appliquée à des programmes de grande taille, la génération de tests, peut manquer de temps ou d'espace mémoire avant de confirmer certaines alarmes comme de vraies erreurs ou conclure qu'aucun chemin d'exécution ne peut atteindre l'état d'erreur de certaines alarmes et donc rejeter ces alarmes. Pour surmonter ce problème, nous proposons de réduire la taille du code source par le *slicing* avant de lancer la génération de tests. Le *slicing* transforme un programme en un autre programme plus simple, appelé *slice*, qui est équivalent au programme initial par rapport à certains critères.

Quatre utilisations du *slicing* sont étudiées. La première utilisation est nommée *all*. Elle consiste à appliquer le *slicing* une seule fois, le critère de simplification étant l'ensemble de toutes les alarmes du programme qui ont été détectées par l'analyse statique. L'inconvénient de cette utilisation est que la génération de tests peut manquer de temps ou d'espace et les alarmes les plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes. Dans la deuxième utilisation, nommée *each*, le *slicing* est effectué séparément par rapport à chaque alarme. Cependant, la génération de tests est exécutée pour chaque programme et il y a un risque de redondance d'analyse si des alarmes sont incluses dans d'autres slices.

Pour pallier ces inconvénients, nous avons étudié les dépendances entre les alarmes et nous avons introduit deux utilisations avancées du *slicing*, nommées *min* et *smart*, qui exploitent ces dépendances. Dans l'utilisation *min*, le *slicing* est effectué par rapport à un ensemble minimal de sous-ensembles d'alarmes. Ces sous-ensembles sont choisis en fonction de dépendances entre les alarmes et l'union de ces sous-ensembles couvre l'ensemble de toutes les alarmes. Avec cette utilisation, on a moins de *slices* qu'avec *each*, et des *slices* plus simples qu'avec *all*. Cependant, l'analyse dynamique de certaines *slices* peut manquer de temps ou d'espace avant de classer certaines alarmes, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple permettrait de les classer. L'utilisation *smart* consiste à appliquer l'utilisation précédente itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire. Lorsqu'une alarme ne peut pas être classée par l'analyse dynamique d'une slice, des slices plus simples sont calculées.

Nous prouvons la correction de la méthode proposée. Ces travaux sont implantés dans SANTE, notre outil qui relie l'outil de génération de tests PATHCRAWLER et la plate-forme d'analyse statique FRAMA-C. Des expérimentations ont montré, d'une part, que notre combinaison est plus performante que chaque technique utilisée indépendamment et, d'autre part, que la vérification devient plus rapide avec l'utilisation du *slicing*. De plus, la simplification du programme par le *slicing* rend les erreurs détectées et les alarmes restantes plus faciles à analyser.

Mots-clés: Analyse statique, slicing, test structurel, erreur à l'exécution, génération de tests guidée par les alarmes, bogue, fausse alarme.

Abstract

Software validation remains a crucial part in software development process. Two major techniques have improved in recent years, dynamic and static analysis. They have complementary strengths and weaknesses.

We present in this thesis a new original combination of these methods to make the research of runtime errors more accurate, automatic and reduce the number of false alarms. We prove as well the correction of the method. In this combination, static analysis reports alarms of runtime errors some of which may be false alarms, and test generation is used to confirm or reject these alarms.

When applied on large programs, test generation may lack time or space before confirming out certain alarms as real bugs or finding that some alarms are unreachable. To overcome this problem, we propose to reduce the source code by program slicing before running test generation. Program slicing transforms a program into another simpler program, which is equivalent to the original program with respect to certain criterion.

Four usages of program slicing were studied. The first usage is called *all*. It applies the slicing only once, the simplification criterion is the set of all alarms in the program. The disadvantage of this usage is that test generation may lack time or space and alarms that are easier to classify are penalized by the analysis of other more complex alarms. In the second usage, called *each*, program slicing is performed with respect to each alarm separately. However, test generation is executed for each sliced program and there is a risk of redundancy if some alarms are included in many slices.

To overcome these drawbacks, we studied dependencies between alarms on which we base to introduce two advanced usages of program slicing : *min* and *smart*. In the *min* usage, the slicing is performed with respect to subsets of alarms. These subsets are selected based on dependencies between alarms and the union of these subsets cover the whole set of alarms. With this usage, we analyze less slices than with *each*, and simpler slices than with *all*. However, the dynamic analysis of some slices may lack time or space before classifying some alarms, while the dynamic analysis of a simpler slice could possibly classify some. Usage *smart* applies previous usage iteratively by reducing the size of the subsets when necessary. When an alarm cannot be classified by the dynamic analysis of a slice, simpler slices are calculated.

These works are implemented in SANTE, our tool that combines the test generation tool PATHCRAWLER and the platform of static analysis FRAMA-C. Experiments have shown, firstly, that our combination is more effective than each technique used separately and, secondly, that the verification is faster after reducing the code with program slicing. Simplifying the program by program slicing also makes the detected errors and the remaining alarms easier to analyze.

Keywords: static analysis, program slicing, all-paths test generation, runtime errors, alarm-guided test generation, bugs, false positive.