THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *15/12/2016* **par :**

Vincent Mussot

Automates d'annotation de flot pour l'expression et l'intégration de propriétés dans l'analyse de WCET

JURY

Jean-Paul BODEVEIX	Professeur d'Université	Président du jury
Hugues CASSÉ	Maître de Conférences	Directeur de thèse
Matthieu MARTEL	Professeur d'Université	Rapporteur
Isabelle PUAUT	Professeur d'Université	Rapporteur
Pascal RAYMOND	Chargé de Recherche	Examinateur
Pascal SOTIN	Maître de Conférences	Co-encadrant

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeur(s) de Thèse :
Hugues CASSÉ et *Pascal SOTIN*

Rapporteurs :
Isabelle PUAUT et *Matthieu MARTEL*

À mon fils.

Automates d'annotation de flot pour l'expression et l'intégration de propriétés dans l'analyse de WCET

Vincent Mussot

Résumé

Dans le domaine des systèmes critiques, l'analyse des temps d'exécution des programmes est nécessaire pour planifier et ordonnancer au mieux différentes tâches et par extension pour dimensionner les systèmes. La durée d'exécution d'un programme dépend de divers facteurs comme ses entrées ou le matériel utilisé. Or cette variation temporelle pose problème dans les systèmes temps-réel dans lesquels il est nécessaire de dimensionner précisément les temps processeur alloués à chaque tâche. Ce point requiert d'être capable d'estimer au mieux le temps d'exécution au pire cas, ou *Worst-Case Execution Time* (WCET) de chaque tâche.

Au sein de l'équipe TRACES à l'IRIT, nous cherchons à calculer une borne supérieure à ce WCET qui soit la plus précise possible. Pour cela, nous travaillons sur le graphe de flot de contrôle (CFG) d'un programme qui représente un sur-ensemble des ses exécutions possibles. Une exécution du programme correspond à un chemin de l'entrée de ce graphe à la sortie. Les chemins peuvent emprunter des boucles dans le graphe mais seulement un nombre fini de fois, car pour estimer le WCET d'une tâche, il est nécessaire que celle-ci se termine. Ainsi, il est primordial de borner les boucles présentes dans le programme, ce qui nous permet lors de notre analyse d'estimer un WCET qui ne soit pas infini. De plus, il est possible d'intégrer dans l'analyse des informations supplémentaires qu'on appellera *annotations* ou *proprétés sémantiques*, qui renseignent sur des comportements spécifiques du programme susceptibles de réduire la sur-approximation de notre estimation.

Dans le domaine du calcul de WCET, les outils utilisent habituellement un langage d'annotation pour exprimer les bornes de boucles et les différentes propriétés sémantiques d'un programme, et pour les intégrer dans l'analyse qui aboutit au WCET estimé. Nous proposons d'utiliser des automates appelés automates d'annotation de flot, ou *Flow Fact Automata* (FFA), pour exprimer les différentes informations habituellement supportées par les langages d'annotation. L'intégration de ces annotations dans l'analyse de WCET pourra ainsi reposer sur la base formelle qu'offre le domaine des automates, et elle pourra être rendue automatique par l'utilisation du produit d'au-

tomates entre des FFAs porteurs d'annotations et un automate équivalent au CFG du programme analysé.

Dans le cadre de cette thèse, nous avons développé des automates enrichis avec des contraintes, des variables et une hiérarchie, afin d'obtenir une expressivité suffisante pour représenter des bornes de boucles, des contextes d'exécution ainsi que l'ensemble des propriétés sémantiques utilisées dans les annotations liées au domaine du WCET.

Nous avons ensuite étendu les opérations existantes sur les automates et défini les nouvelles opérations nécessaires pour mener à bien l'idée de faire un produit avec un automate équivalent au CFG dont le résultat intégrerait les différentes informations initialement portées par les automates.

Par la suite, ces différents éléments ont été intégrés dans un contexte d'analyse de WCET : sur la base du framework Otawa développé dans notre équipe, nous avons mis en place une génération d'automates à partir du langage d'annotation FFX pour chaque type de balise utilisées actuellement. Puis avec l'aide des différentes opérations définies, nous avons intégré les informations portées par les automates dans l'analyse de WCET.

Parallèlement, nous avons proposé une nouvelle solution issue de la nature des automates enrichis pour améliorer la précision de notre analyse de WCET. L'intégration des annotations dans une analyse se fait habituellement par l'association de contraintes numériques au graphe de flot de contrôle. Les automates que nous présentons supportent cette méthode mais leur expressivité offre également de nouvelles possibilités d'intégration basées sur un dépliage partiel du CFG. Nous avons mis à l'épreuve cette méthode par des expérimentations et par la comparaison avec la méthode habituellement utilisée.

Pour finir, nous avons isolé et spécifié une catégorie de propriétés sémantiques que nous avons définie dans le langage d'annotation FFX, et nous avons développé les automates capables de supporter cette nouvelle classe de propriétés. Cette catégorie se prêtant bien à la comparaison de la nouvelle forme d'intégration introduite par les automates, nous avons pu vérifier l'efficacité de notre approche face à celle existante au travers d'une amélioration de la précision du WCET estimé dans diverses expérimentations.

Mots-Clés : WCET, systèmes temps-réel, analyse statique, Automates hierarchiques, IPET, langage d'annotation

Flow Fact Automata for the Expression and the Integration of Properties in WCET Analysis

Vincent Mussot

Abstract

In the domain of critical systems, the analysis of execution times of programs is needed to schedule various task at best and by extension to dimension the whole system. The execution time of a program depends on multiple factors such as its entries or the targeted hardware. This time variation is an issue in real-time systems where the duration is required to allocate correct processor time to each task, and in this purpose, we need to know their worst-case execution time.

In the TRACES team at IRIT, we try to compute a safe upper bound of this worst-case execution time that would be as precise as possible. In order to do so, we work on the control flow graph of a program that represents an over-set of its possible executions and we combine this structure with annotations on specific behaviours of the program that might reduce the over-approximation of our estimation.

Tools designed to compute worst-case execution times of programmes usually support the expression and the integration of annotations thanks to specific annotation languages. Our proposal is to replace these languages with a type of automata named flow fact automata so that not only the expression but also the integration of annotations in the analysis inherit from the formal basis of automata. We show how these automata enriched with constraints, variables and a hierarchy support the various annotations used in the worst-case execution time domain.

Additionally, the integration of annotations in an analysis usually relies on the association of numerical constraint to the control flow graph. The automata presented here support this method but their expressiveness offers new integration possibilities based on the partial unfolding of the control flow graph. We present experimental results from the comparison of these two methods that show how the graph unfolding can improve the analysis precision. In the end, this precision gain in the worst-case execution time will ensure a better usage of the hardware as well as the absence of risks for the user or the system itself.

Keywords : WCET, real-time systems, static analysis, IPET, hierarchical automata, annotation language.

Table des matières

Table des figures

Liste des tableaux

Liste des codes

"Science is neat, but I'm afraid it's not very forgiving"

— Scott Clarke

1 Introduction

Dans le domaine des systèmes embarqués, l'analyse des temps d'exécution des programmes est nécessaire pour planifier et ordonnancer au mieux différentes tâches et par extension pour dimensionner les systèmes. Dans un contexte temps-réel strict où chaque programme doit respecter sa date limite, planifier les tâches en s'appuyant sur la durée maximale d'exécution de chacune d'elles permet d'assurer le fonctionnement correct du système tout en limitant ses dimensions.

La durée maximale d'exécution d'un programme est appelée *temps d'exécution au pire cas*, ou *Worst-Case Execution Time* (WCET). En théorie, écrire un algorithme capable de calculer ce WCET est impossible en raison des conditions et des imbrications de boucles des programmes qui provoquent une explosion exponentielle du nombre de cas à tester. En pratique, nous nous efforçons de calculer une sur-estimation de ce WCET qui soit la plus précise possible. Une méthode consiste à travailler sur le Graphe de Flot de Contrôle (CFG) du programme qui représente un sur-ensemble de ses exécutions possibles. Une exécution du programme correspond à un chemin de l'entrée à la sortie de ce graphe, mais la présence de cycles dans le graphe n'interdit pas les exécutions infinies pour lesquelles le temps d'exécution est lui aussi infini. De

fait, l'estimation du WCET n'a de sens que sur l'ensemble des exécutions qui se terminent. Il est donc nécessaire de connaître les bornes des boucles présentes dans le programme et de les associer au CFG pour supprimer les chemins correspondants à ces exécutions infinies. Ces chemins qui ne correspondant à aucune exécution réelle du programme sont appelés des *chemins infaisables*, dont la présence provoque un risque de sur-estimation de WCET. Il est donc souhaitable d'en supprimer le plus grand nombre possible en intégrant dans l'analyse des informations supplémentaires qu'on appellera des *propriétés sémantiques* ou des *annotations*.

Dans le domaine de l'analyse de WCET, les outils utilisent habituellement un *langage d'annotation* pour exprimer ce type d'informations et les intégrer dans leur estimation du WCET. Nous proposons d'utiliser des automates qu'on appellera *automates d'annotation de flot* pour supporter les différentes informations sémantiques et les bornes de boucles, et de se reposer sur le formalisme existant dans le domaine des automates pour intégrer ces informations dans le CFG.

Par ailleurs, les langages d'annotation ont souvent été définis en lien avec une méthode d'analyse de WCET précise. Dans le cas de la méthode d'estimation du WCET par énumération implicite de chemins (IPET), l'intégration d'annotations de flot dans l'analyse se limite généralement à l'ajout de nouvelles contraintes dans le système d'équations associé à la méthode. L'expressivité des automates offre des perspectives différentes puisqu'elle permet une intégration des annotations par dépliage partiel du CFG, ce qui permet de gagner en précision dans l'estimation du WCET au prix d'une complexification de l'analyse.

Organisation du Document

Ce mémoire de thèse s'architecture de la façon suivante :

Le chapitre 2 présente l'état de l'art relatif aux analyses permettant d'estimer le WCET d'un programme. Diverses méthodes sont mentionnées et l'expression et l'intégration des propriétés sémantiques dans l'analyse de WCET sont détaillées. Nous concluons ce chapitre en mentionnant les limitations des langages d'annotations. Nous justifions ainsi la recherche d'un formalisme dont l'expressivité offrirait de nouvelles façon d'intégrer les annotations dans l'analyse. La flexibilité de la méthode d'intégration envisagée fera ainsi apparaître le potentiel d'amélioration de la précision de l'analyse.

Dans le chapitre 3, nous présentons les automates d'annotations comme support à l'expression des propriétés sémantiques. Nous montrons comment ces automates enrichis de contraintes peuvent se substituer à un langage d'annotation et s'intégrer efficacement dans un processus d'analyse de WCET. Il ressort de ce chapitre la nécessité d'étendre le formalisme utilisé à une hiérarchie d'automates afin d'obtenir une expressivité au moins équivalente à celle d'un langage d'annotation.

Le chapitre 4 introduit une structure hiérarchique dans les automates d'annotation avec pour objectif de représenter les différents contextes qu'on trouve habituellement dans la programmation impérative. Nous détaillons également comment elle peut s'intégrer dans une analyse de WCET et nous présentons les différents algorithmes développés à cette fin.

Le chapitre 5 s'appuie sur un langage d'annotation existant (FFX) pour montrer l'utilisation des automates d'annotation. Nous illustrons comment il est possible de générer des automates à partir des différentes annotations communément utilisées dans le domaine du calcul de WCET.

Dans le chapitre 6, nous montrons comment il est possible d'exploiter la flexibilité des automates pour faire varier la précision des résultats de l'analyse de WCET. Nous présentons une méthode reposant sur un dépliage de graphe comme alternative à la méthode traditionnelle d'intégration par ajout de contraintes. Nous observons ensuite les effets du compromis entre ces deux méthodes d'intégration et les différences de précision de l'analyse de WCET qui peuvent en ressortir. Nous présentons enfin une méthode systématique pour transformer certaines contraintes en un automate capable de déplier partiellement le CFG du programme analysé, tout en conservant la possibilité d'intégration sous forme de contrainte.

Dans le chapitre 7, nous nous concentrons sur une nouvelle catégorie d'annotations relative à un type de chemins infaisables que certains outils sont en mesure de fournir. Nous exprimons ces annotations en reprenant les deux types d'automates présentés dans le chapitre 6, mais pour surmonter les problèmes de complexité liés au dépliage partiel du CFG, nous dérivons une propriété d'ordre entre les éléments d'une même annotation. Il nous est alors possible de comparer l'intégration par les deux types d'automates à partir de suites programmes fréquemment utilisées dans l'analyse de WCET.

Nous concluons cet ouvrage en mettant en avant les bénéfices qui peuvent être tirés de l'utilisation des automates d'annotations dans l'analyse de WCET. Nous appliquons

notamment une grille de critères relative aux langages d'annotation pour estimer la valeur de cette proposition, dont nous pointons également les limitations. Enfin, nous fournissons des pistes diverses pour améliorer notre méthode ou l'étendre à des voies de recherches inexplorées.

"C'est pas faux !"

— Perceval le Gallois

2

État de l'art

Sommaire

2.1 Introduction

La propriété principale qu'on attend généralement d'un programme est qu'il soit correct, c'est-à-dire que ses sorties correspondent bien à celles attendues par sa spécification. Dans le domaine des systèmes temps-réel cependant, l'exécution d'un programme se doit également de respecter des contraintes temporelles.

Considérons un ordonnanceur qui alloue un créneau de temps processeur à une

tâche pour qu'elle puisse s'exécuter. Afin que son exécution se déroule correctement, ce créneau de temps doit être suffisamment grand pour que la tâche puisse se terminer, et ce, quelles que soient ses entrées. Si ce n'est pas le cas, il est possible que la tâche dépasse le créneau de temps qui lui était alloué, c'est-à-dire qu'elle ne respecte pas sa date limite (*deadline*) d'utilisation du processeur. En fonction du type d'ordonnanceur utilisé, les risques seront soit la non-terminaison de la tâche courante à cause de la préemption d'une nouvelle tâche (ré-allocation du processeur), ou à l'inverse le décalage, voire le report de la tâche suivante qui n'aura pas pu démarrer à temps.

On parlera de système temps-réel strict quand ce non-respect de *deadline* par une tâche est susceptible d'avoir un impact critique sur le bon fonctionnement du système, avec des conséquences potentiellement catastrophiques pouvant aller jusqu'à menacer des vies humaines. Les systèmes impliquant des contraintes temps-réel strictes s'étendent du domaine médical (les *pacemakers* [58] par exemple) aux domaines aéronautique et automobile (les pilotes automatiques, les contrôleurs de vol, ...), en passant par la robotique, les systèmes de surveillance d'usines, etc.

Ainsi, dans les systèmes critiques, on s'attache à allouer à une tâche précise un créneau de temps processeur qui soit adapté, de façon à ce qu'elle soit en capacité de respecter sa date limite. Pour y parvenir, une méthode consiste à calculer son temps d'exécution au pire cas ou *Worst-Case Execution Time* (WCET). En effet, la durée d'une tâche peut varier en fonction de ses entrées par exemple, ou du matériel sur lequel la tâche est exécutée. Un observateur omniscient capable de passer en revue toutes les exécutions possibles d'une tâche pourrait aisément trouver celle dont la durée est la plus longue. Néanmoins, il est théoriquement impossible d'écrire un algorithme qui puisse jouer le rôle de cet observateur omniscient puisque la preuve de la terminaison d'un programme est déjà un problème indécidable en soi [66]. On utilisera donc, à la place, des méthodes capables de fournir soit une borne garantie supérieure ou égale à ce WCET, soit un WCET approché issu d'une mesure et parfois associé à un indice de confiance [1].

La figure 2.1 montre les durées de toutes les exécutions possibles d'une tâche du point de vue d'un observateur omniscient. Ce graphique illustre le comportement d'un programme dont la majeure partie des exécutions est bien inférieure au WCET réel mais pour lequel quelques cas spécifiques déclenchent des traitements plus longs (typi-

1. Pourcentage d'exécutions pour lesquelles le WCET estimé sera bien respecté. Par exemple : "Le WCET sera inférieur à 1 ms dans 95% des cas"

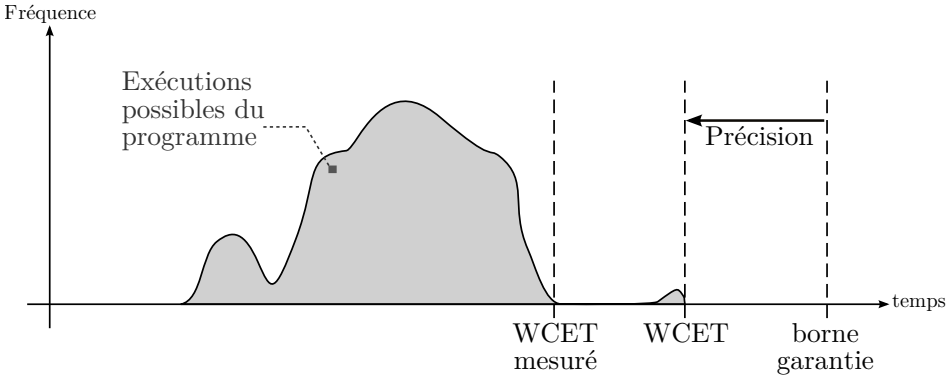


Figure 2.1 – Distribution de la durée de toutes les exécutions d'un programme.

quement des cas d'erreurs associés à des gestions d'exceptions). D'un côté, on trouve une borne garantie supérieure au WCET réel. La distance entre cette borne et le WCET correspond à la précision avec laquelle la borne a pu être estimée. De l'autre côté, on trouve le WCET mesuré, qui correspond au maximum de la majeure partie des exécutions.

Après une introduction des notions de base utilisées dans cette thèse et relatives à la représentation des programmes, nous présenterons les différentes approches existantes permettant d'estimer le WCET et les principales méthodes utilisées. Nous présenterons également des moyens de modéliser le matériel, ainsi que les structures et langages permettant d'aboutir à une estimation du WCET.

Représentation des programmes

Nous utiliserons dans cette thèse la représentation des programmes sous forme de graphes de flot de contrôle ou *Control Flow Graphs* (CFGs) [2]. Cette représentation constitue un sur-ensemble des exécutions possibles du programme et repose sur le découpage du code assembleur correspondant au code machine en blocs de base qui sont définis ainsi :

Définition 1. *Un bloc de base est une séquence maximale d'instructions dont la première instruction est le seul point d'entrée et dont la dernière instruction est le seul point de sortie.*

Un CFG est donc composé de tous les blocs de base qui composent un programme et d'arcs les reliant. Ces arcs traduisent la possibilité de passer d'un bloc à l'autre, soit en séquence quand la dernière instruction d'un bloc de base et la première instruction

d'un second bloc ont des adresses qui se suivent, soit par une instruction spécifique de branchement. On définit un CFG ainsi :

Définition 2. *Le CFG d'un programme est un graphe orienté G = (B,E,e,x) tel que B est un ensemble de blocs de base* $\{b_1, b_2, \ldots, b_n\}$*, E est un ensemble d'arêtes orientées* $\{(b_i, b_j), (b_k, b_l), \ldots\}$*, e est une unique entrée et x une unique sortie. Chaque arête est une paire ordonnée de blocs* (b_i, b_j) *où* b_i *est la source de l'arête et* b_j *la destination. On exige que pour tout bloc de base* $n \in B$*, n soit atteignable depuis e, et x soit atteignable depuis n.*

Le CFG est un graphe orienté, pour lequel on peut définir la notion de chemin :

Définition 3. *Un chemin dans un CFG G = (B,E,e,x) est une séquence de blocs de base* $(b_1, b_2, \ldots, b_n) \in B^\star$*, avec* $b_1 = e$*,* $b_n = x$ *et* $\forall i \in [1, n-1], (b_i, b_{i+1}) \in E$*.*

Cette notion de chemin est à mettre en correspondance avec les exécutions possibles du programme : une exécution du programme correspond à un chemin dans le CFG (mais l'inverse n'est pas toujours vrai).

2.2 Approches générales de l'analyse de programmes

Intuitivement, analyser un programme pour en extraire une propriété (son WCET par exemple) semble aller de pair avec son exécution. Cette approche appelée *analyse dynamique* est effectivement utilisée, mais on l'oppose généralement à une seconde approche nommée *analyse statique*, qui consiste à étudier un programme et en dériver des propriétés sans l'exécuter. Cette partie présente ces deux catégories en considérant comme objectif le calcul du WCET.

2.2.1 Analyse dynamique

Cette approche appelée *measurement based method* [59] consiste à exécuter plusieurs fois un même programme, au moyen d'un jeu de tests, en mesurant à chaque fois sa durée d'exécution. Si un des tests effectués correspond bien à l'exécution au pire cas du programme, cette méthode fournira alors son WCET exact. Néanmoins, le seul moyen d'assurer que cette exécution au pire cas sera bel et bien rencontrée est de tester toutes les entrées possibles du programme, ce qui est possible sur des programmes

triviaux, mais inenvisageable en pratique. À la place, on utilise généralement des tests supposés représentatifs [52], c'est-à-dire qu'ils sont susceptibles de permettre de trouver le WCET, mais sans pour autant le garantir. Pour tenter de palier ce problème, la pratique dans l'industrie consiste généralement à ajouter une marge de sécurité au WCET mesuré. Cependant le problème demeure puisque d'une part, cela ne garantit toujours pas que le WCET réel ait bien été dépassé, et d'autre part, cela aboutit généralement à une sur-estimation importante.

Un autre point faible de cette approche, est que certaines méthodes destinées à mesurer le temps d'exécution d'un programme utilisent l'instrumentation du code pour y intégrer les mesures, ce qui modifie son exécution. Les temps mesurés sont donc directement impactés par les mesures et les résultats reportés sur le programme initial sont forcément biaisés (sur-estimés en général). Ce point n'est cependant pas critique puisqu'il existe également des méthodes de mesure non intrusive qui ne modifieront pas le programme, mais qui reposent généralement sur les spécificités matérielles de la machine cible. L'outil RapiTime [54] par exemple supporte diverses méthodes intrusives et non-intrusives pour effectuer ces mesures.

2.2.2 Analyse statique

Le principe de l'analyse statique repose sur l'extraction des propriétés d'un programme sans l'exécuter. Cette méthode est prisée dans le domaine du WCET car elle a l'avantage de fournir une borne garantie supérieure au WCET réel du programme. Par exemple, pour un programme donné, l'analyse statique considère toutes les entrées possibles et se base sur une modélisation du système cible pour estimer le WCET du programme. Pour palier l'explosion combinatoire qui découlerait de cette tâche si on calculait exhaustivement le WCET pour chaque entrée possible, on utilise une méthode inventée par Patrick et Radhia Cousot appelée *interprétation abstraite* [14] ou *Abstract Interpretation* (AI). Cette méthode consiste à définir une abstraction à partir d'un domaine concret en se basant sur un formalisme qui assure en retour la conservation des propriétés découvertes sur le domaine abstrait vers le domaine concret. L'utilisation d'intervalles à la place de valeurs concrètes par exemple constitue une abstraction.

L'interprétation abstraite permet ainsi de réduire la complexité des modèles et des différentes analyses tout en garantissant les propriétés dérivées des domaines abstraits. On la retrouve à diverses étapes de l'estimation de WCET par analyse statique, en

particulier dans la définition de modèle abstrait de l'architecture matérielle.

On présente dans la partie suivante les différentes méthodes permettant d'estimer une borne garantie au WCET par cette approche statique.

2.3 Méthodes d'estimation du WCET

On commencera cette partie par la présentation des méthodes existantes dans le domaine du calcul de WCET avant de se concentrer sur la méthode dans laquelle s'intègrent les travaux présentés dans cette thèse. Cette dernière méthode repose sur l'association de l'*interprétation abstraite* pour la modélisation de l'architecture et sur la méthode d'énumération implicite de chemins ou *Implicit Path Enumeration Technique* (IPET) pour l'estimation du WCET.

2.3.1 Tree-based

La méthode Tree-based introduite dans l'article [51], aussi appelée *Timing Schema* [48, 61] est fondée sur la structuration arborescente des programmes analysés.

L'idée est de représenter le programme sous forme d'arbre, dont les nœuds correspondent aux structures de contrôle du programme (conditions, boucles, appels de fonctions...) et dont les feuilles correspondent aux morceaux de code qu'on trouve entre ces éléments de contrôle. On représente ainsi le programme selon les imbrications de ses différents contextes, et non pas sous une forme classique de CFG car celui-ci n'offre pas la structure arborescente qui sert de fondement à cette méthode.

L'étape suivante consiste à calculer séparément les temps d'exécution de chaque feuille qui correspondent aux morceaux de codes du programme. Reste ensuite à remonter les temps calculés vers la racine de l'arbre pour obtenir le WCET du programme complet, mais pour cela, il est nécessaire de combiner les différents temps calculés en définissant des règles pour chaque structure de contrôle rencontrée : pour une condition par exemple, on prendra le temps maximum des deux branches, pour une boucle, on multipliera le temps calculé dans la boucle par sa borne, etc.

Ces travaux préliminaires sont basés sur l'indépendance des instructions entre différents contextes, or cette indépendance est remise en question sur les processeurs modernes à cause des effets de l'utilisation de caches, de l'exécution de plusieurs ins-

tructions en parallèle grâce aux pipelines des processeurs, ou encore de la prédiction de branchement. Des efforts ont été faits pour intégrer ces éléments dans la méthode [12] mais des limitations persistent, notamment l'hypothèse d'une forte correspondance entre la structure du code source et celle du binaire, ce qui limite le support des optimisations durant la compilation. En outre, comme mentionné dans [70] et [30], cette approche supporte mal l'ajout d'informations de flot additionnelles comme cela peut être fait dans d'autres méthodes.

2.3.2 Path-based

La méthode Path-based [62] consiste à modéliser explicitement chaque chemin du CFG pour trouver par une recherche exhaustive le chemin dont la durée sera la plus longue, qui correspondra alors au WCET estimé. Cette méthode est efficace sur des programmes simples avec peu de chemins, ou dans des contextes spécifiques comme à l'intérieur d'une boucle. Toutefois, l'imbrication des boucles, ou le fait que le nombre de chemins d'un programme augmente de façon exponentielle avec les branchements qui le composent mettent à mal cette méthode exhaustive.

Si cette méthode souffre d'un problème de complexité, elle est efficace sur des programmes composés de peu de chemins. Cette observation a motivé les travaux présentés dans l'article [50] qui s'intéressent à la génération de programmes comportant un chemin d'exécution unique à partir de programmes traditionnels, afin de pouvoir utiliser cette approche Path-based efficacement et éviter les problèmes de complexité. Néanmoins, les codes générés ainsi souffrent de mauvaises performances temporelles (temps d'exécutions supérieur au WCET du programme initial) ce qui les rend globalement inefficaces.

2.3.3 Model Checking

Dans l'article [69], il est dit que le *model checking* n'est pas adapté à l'analyse de WCET, mais cette affirmation est contredite dans l'article [42]. Celui ci présente une modélisation du comportement du cache basée sur une sémantique issue des automates et intégrant les informations de flot du programme, à laquelle est associée une méthode de calcul de WCET fondée sur le *model checking*.

Généralement, l'estimation de WCET par *model checking* est divisée en deux étapes.

La première consiste à modéliser la durée du programme en associant à une représentation de ses flots d'exécution des horloges ou des coûts correspondant à la durée écoulée depuis l'entrée du programme. La seconde étape est la recherche du WCET à proprement parler, qui consiste à fournir au *model checker* un temps supposé supérieur au WCET, et de vérifier si les temps de sortie de l'outil de modélisation sont tous inférieurs à ce temps supposé. En cas de succès, ce temps est considéré comme une borne, et on définit par dichotomie un nouveau temps supposé qui sera fourni au *model checker*.

On retrouve à l'heure actuelle des estimations de WCET par *model checking* dans divers articles [15, 68, 5, 10], qui utilisent généralement le model checker UPPAAL [6] et une modélisation par des *timed automata* [3].

2.3.4 Combiner AI et IPET

Dans l'article [69], les principales méthodes utilisées dans le domaine de l'analyse de WCET sont comparées et il en ressort que la combinaison la plus prometteuse consiste à utiliser l'interprétation abstraite pour modéliser l'architecture et la méthode IPET pour estimer le WCET. Par ailleurs, même si l'analyse de WCET par *model checking*, rejetée par ce même article, a depuis été réhabilitée (voir partie 2.3.3), il n'en reste pas moins qu'associer AI et IPET est devenue une méthode de référence pour l'estimation du WCET par analyse statique. Notons qu'un des points fort de la méthode est sa flexibilité en terme d'intégration d'informations de flot additionnelles dans l'analyse, ce qui a joué un rôle dans sa prédominance sur les autres méthodes comme Tree-based.

On détaillera dans cette partie l'ensemble du processus de l'analyse statique permettant d'aboutir au calcul du WCET et que l'on sépare généralement en trois étapes :

- La première appelée *analyse de flot* consiste à enrichir le CFG d'un programme avec des informations supplémentaires pour avoir la vision la plus précise possible de l'ensemble des exécutions du programme sur lesquelles on appliquera notre estimation de WCET.
- La seconde consiste à modéliser l'architecture et à estimer les temps des blocs de bases et des différents éléments matériels qui peuvent être pris en compte dans l'analyse de WCET, comme les caches, les effets du pipeline, etc.
- Enfin, la troisième repose sur la méthode IPET associée à une méthode de résolution de contraintes pour trouver le chemin (ou l'ensemble de chemins) dans

le CFG qui correspond à l'exécution la plus longue possible en prenant en compte les différents temps estimés lors la seconde étape et les informations de flot issues de la première.

2.3.4.1 L'analyse de flot

Le CFG (définition 2) est généralement le point de départ de cette analyse puisqu'il constitue déjà un sur-ensemble des exécutions possible du programme analysé. De fait, il existe forcément un chemin dans le CFG qui aboutit au WCET réel du programme. Néanmoins, il existe également des chemins dans le CFG qui ne correspondent à aucune exécution possible du programme, qu'on appellera des *chemins infaisables*.

Par ailleurs, il existe un chemin appelé chemin d'exécution au pire cas ou *Worst-Case Execution Path* (WCEP) qui correspond à la plus longue durée d'exécution qu'il est possible de trouver dans le CFG. Si l'on fait l'hypothèse d'une analyse omnisciente capable découvrir ce WCEP et que celui-ci est un chemin faisable du graphe, alors sa durée d'exécution est égale au WCET réel. En revanche, si ce chemin au pire cas est infaisable, cela signifie que le chemin correspondant au WCET réel a une durée inférieure à celle du WCEP courant (cette durée constitue donc une borne du WCET). Ces chemins infaisables sont une des sources de la sur-estimation du WCET par analyse statique.

Le but de l'analyse de flot est donc de réduire au maximum la présence de chemins infaisables dans le CFG, en fournissant des informations additionnelles. Par exemple, si un programme contient des boucles, on trouvera alors des cycles dans le CFG, et en l'absence de bornes pour les boucles, le WCEP correspondra à une exécution infinie. Il est donc nécessaire d'associer les bornes des boucles au CFG du programme pour être en mesure d'estimer un WCET. Ce genre de propriétés externes, peuvent être fournies soit par le développeur lorsqu'il en est capable, soit par une analyse automatique. On parlera généralement d'*annotations* pour ces propriétés externes et d'*annotations de flot* lorsqu'elles concerneront des informations sur les flots d'exécution d'un programme (qui correspondent aux arcs de son CFG).

Les annotations de flot : Dans le domaine de l'estimation de WCET, les annotations de flot qui sont habituellement fournies et utilisées sont les suivantes :

- Des contraintes permettant de restreindre la fréquence d'exécution d'arcs spécifiques

du CFG.

- Des dépendances entre les exécutions de différents arcs du CFG (pour les conditions par exemple).
- Les bornes des boucles qui limitent le nombre d'itérations des boucles. Ces bornes peuvent également être exprimées par des contraintes de fréquence d'exécution de certains arcs (typiquement les arcs retour des boucles) mais certaines analyses capables de transformer les informations de flot en fonction des optimisations de compilation [30] requièrent de conserver ces informations de bornes de boucles séparées des arcs du CFG.
- Des informations additionnelles sur les boucles : les relations entre les nombres d'itérations de deux boucles imbriquées par exemple, le nombre total des exécutions d'une boucle [43] ou encore des contraintes sur les fréquences d'exécution d'arcs pour des sous-ensembles d'itérations d'une boucle [18].

En outre, d'autres annotations peuvent être fournies à l'analyse comme des informations sur les étendues de valeurs d'entrée du programme [22] mais elles nécessitent d'être transformées en informations de flot pour pouvoir être prises en compte dans l'estimation de WCET.

Les annotations fournies par un expert seront généralement relatives au code source du programme, pour des raisons de simplicité. Il est donc généralement difficile de retranscrire ces annotations sur le code binaire du programme à cause de l'étape de compilation et plus particulièrement lorsque des options d'optimisation sont utilisées. Des efforts ont cependant été faits pour automatiser l'adaptation des annotations lors de compilations optimisées [30, 37].

Par ailleurs, des analyses spécifiques sont également capables de générer automatiquement des annotations, notamment pour les bornes de boucles. Les recherches dans ce domaine utilisent diverses méthodes comme le *pattern matching* [25], souvent combiné avec de l'interprétation abstraite [22, 24, 65].

Le format pour les annotations : Certains langages de programmation spécifiques comme WCETC [29] permettent de spécifier des informations de flot directement dans le code source, mais d'une manière générale, les annotations sont fournies à l'analyse dans des fichiers séparés, et exprimées dans un langage d'annotation spécifique souvent associé à l'outil d'analyse ciblé. Une vue d'ensemble des informations supportées par ces langages d'annotation est présentée dans la partie 2.4.

2.3.4.2 Modéliser l'architecture par interprétation abstraite

Dans les méthodes d'estimation du WCET par analyse statique basées sur le CFG, le principe général est de trouver le chemin dans le graphe correspondant à l'exécution la plus longue du programme, avec comme point de départ la durée d'exécution de chaque bloc de base. Pour obtenir ces temps sans exécuter le programme, il est nécessaire de modéliser le comportement du matériel pour les différentes instructions de chaque bloc de base, afin d'en dériver leur temps d'exécution.

Pour un processeur simple, utiliser une constante issue du manuel du processeur pour chaque type d'instruction suffit à représenter leur durée, mais l'architecture des processeurs modernes est composée de diverses briques et techniques visant à exécuter les programmes toujours plus rapidement. On trouve par exemple une hiérarchie de mémoires et de caches qui permettent de profiter de la localité des instructions et des données, des pipelines permettant de paralléliser le traitement et l'exécution de plusieurs instructions, ou encore des méthodes de prédiction de branchements pour anticiper les cibles des instructions de branchement.

La modélisation complète et exhaustive de l'architecture d'un processeur n'est pas envisageable en pratique pour des raisons de complexité – certains procédés comme l'exécution dans le désordre (*out-of-order execution*) par exemple posent de sérieux problèmes aux méthodes d'analyse statique [41]. À la place, il suffit d'utiliser des valeurs conservatives pour éviter ces problèmes de complexité. Pour une architecture avec un cache par exemple, on peut considérer que tous les accès à la mémoire provoquent des *miss* dans le cache et entraînent une pénalité de temps qu'on ajoute à la durée d'exécution de l'instruction en elle-même (il conviendra cependant de vérifier l'absence globale de *timing anomalies* [21], comme le fait que considérer uniquement les temps maximum d'exécution locaux n'assure pas toujours de sur-estimer le WCET global).

L'interprétation abstraite permet de modéliser des états abstraits du matériel, plus simples que les états concrets, mais porteurs des propriétés qui nous intéressent dans le WCET.

Une modélisation abstraite des caches par exemple est décrite dans divers articles [63, 64, 69]. Elle a pour but de classifier les accès au cache dans quatre catégorie, selon que l'instruction produira toujours un *hit* du cache lors de l'accès à la mémoire (*always hit*) ou toujours un *miss* du cache (*always miss*) auquel cas il faudra appliquer une pénalité d'accès à la mémoire, selon que l'instruction sera disponible (*persistent*)

en cache après un premier chargement (un unique *miss*), ou qu'il n'est pas possible de classifier cette instruction (*not classified*).

Pour aboutir à cette classification, l'interprétation abstraite est utilisée à partir d'états abstraits du cache :

- Une première analyse appelée *Must analysis* consiste à vérifier que quel que soit le chemin d'exécution emprunté, un bloc mémoire donné est toujours présent dans le cache à un point précis du programme. Les instructions accédant à ce bloc à cet endroit pourront donc être classifiées *always hit*. Cela ne signifie pas cependant qu'un bloc mémoire dont la présence dans le cache n'est pas assurée est absent du cache, mais seulement que l'analyse n'a pas pu certifier sa présence.
- Pour s'assurer de l'absence d'un bloc mémoire dans le cache, ce qui permettra de classifier une instruction en *always miss*, on utilisera l'analyse appelée *May analysis* qui consiste à vérifier qu'il n'existe aucun chemin d'exécution pour lequel le bloc mémoire pourrait encore être dans le cache.
- La troisième analyse appelée *persistence analysis* concerne les boucles du programme, et permet de déduire qu'après la première itération d'une boucle, certains blocs mémoire resteront présents dans le cache jusqu'à la sortie de cette boucle. Les instructions accédant à ces blocs mémoire seront donc classés *persistent*.
- Pour finir, toutes les instructions accédant à la mémoire qui n'auront pas pu être classées seront *not classified*.

L'interprétation abstraite est également utilisée pour représenter d'autres éléments de l'architecture des processeurs comme les pipelines [65, 57] par exemple. Les différentes durées estimées des blocs de base à partir de cette modélisation par AI du matériel seront ensuite combinées dans la méthode IPET d'estimation du WCET.

2.3.4.3 La méthode d'énumération implicite de chemins (IPET)

La méthode IPET consiste à combiner dans un systèmes de contraintes numériques les flots d'exécution d'un programme et les temps d'exécution de ses blocs de base. Pour cela, on associe un coefficient de temps (t_{bloc}) à un bloc du CFG, correspondant à sa durée maximale d'exécution, ainsi qu'un compteur (x_{bloc}) correspondant au nombre de fois où ce bloc de base est exécuté. On exprime ensuite les informations de flot issues du CFG et de l'analyse de flot sous forme de contraintes, en utilisant notamment la loi des nœuds pour retranscrire les contraintes structurelles du CFG.

À partir de ces variables et contraintes, il est possible de déterminer une borne garantie supérieure au WCET du programme en maximisant la somme des produit des compteurs et des temps d'exécution de chaque bloc. On maximisera ainsi la fonction $\sum_{i \in blocs} x_i \times t_i$ qu'on appellera *fonction objectif* en ILP.

On utilise communément la programmation linéaire en nombres entiers ou *Integer Linear Programming* (ILP) [11] pour maximiser cette fonction objectif en respectant les contraintes fournies par la méthode IPET et obtenir ainsi le WCET, et ce pour des raisons de disponibilité et d'efficacité des outils. Il est cependant possible d'utiliser d'autres méthodes comme les méthodes de recherche opérationnelle associées aux problèmes de satisfaction de contraintes par exemple [47, 17].

Historiquement, la méthode IPET a été proposée séparément par Puschner et Schedl dans [53] et par Li et al. dans [38] où elle a reçu son nom, qui retranscrit le fait que le WCEP n'est plus défini explicitement mais implicitement. Li et al. ont étendu leur méthode dans [39, 40] en incluant la modélisation des effets du cache dans les contraintes ILP, mais le résultat souffre d'une complexité exponentielle lors de la résolution du système ILP, comme indiqué dans [69] et [49].

La modélisation du flot de contrôle du programme par un système ILP a cependant été conservée et adoptée par de nombreux groupes de recherche et elle a été plus tard combinée avec l'interprétation abstraite utilisée pour modéliser les effets du matériel [63, 64, 69, 16, 19].

2.4 Les langages d'annotation

Dans [33] et [32], Kirner et al. cherchent à définir de façon exhaustive les critères permettant de classer et d'évaluer la qualité des langages d'annotation et comparent la plupart des langages existants. Il apparaît notamment que les langages sont fortement liés à la méthode d'estimation du WCET qu'ils ciblent, mais peuvent néanmoins être évalués sur la base d'un ensemble de critères communs, dont le plus important est leur expressivité.

Ce critère primordial traduit la capacité d'un langage d'annotation à supporter les différentes informations de flot. La complétude de cette expressivité requiert d'un langage qu'il soit en mesure de décrire précisément tous les chemins faisables d'un programme arbitraire – un tel langage est nommé *path-complete*. En outre, les articles

mentionnent les informations de flot qui sont susceptibles d'être supportées par les langages d'annotation, à commencer par les bornes de boucles qui sont indispensables à l'analyse de programmes comportant des boucles. Les autres annotations de flot mentionnées sont :

1. les bornes des boucles non-rectangulaires [2] (*non-rectangular loops*),
2. la sensibilité au contexte d'appel, c'est-à-dire la possibilité de définir des annotations de flot pour différents appels d'une même fonction,
3. la sensibilité au contexte de boucle, c'est-à-dire la capacité à exprimer des annotations pour un sous-ensemble des itérations d'une boucle donnée,
4. le nombre d'exécution explicite de certains éléments du code source ou du code binaire, ou des relations numériques entre eux,
5. la notion de contexte d'application correspondant à la différenciation d'une même fonction sur la base de ses paramètres d'entrée,
6. l'ordre d'exécution de certains blocs de base ou arcs du CFG.

Parmi les autres critères considérés, on trouve notamment la complexité du langage, son niveau d'abstraction, c'est-à-dire le support d'annotations sur le code source et/ou sur le code binaire, ou encore le placement des annotations (dans le code source ou dans un fichier à part).

Parmi les langages associés à une estimation de WCET par IPET, on peut notamment citer le langage aiS [20] qui supporte toutes les annotations de flot mentionnées, à l'exception du point 6. À l'heure actuelle, le langage FFX supporte également une grande partie de ces annotations de flot à l'exception des points 5 et 6. Nous présentons ce dernier langage dans la partie suivante car il servira de base à une traduction depuis des annotations vers les automates présentés dans cette thèse.

2.4.1 FFX

Fondé sur le langage de balisage extensible ou *Extensible Markup Language* (XML), le langage d'annotation *Flow Fact in XML* (FFX) [8] a été pensé comme un langage portable et convivial et développé dans une optique d'harmonisation des langages d'annotations. Il est supporté par les outils d'analyse de WCET Otawa [4], TuBound [36],

2. Boucles imbriquées dont la borne de la boucle interne dépend de l'itération de la boucle externe

CalcWCET167 [31] et par l'outil de génération d'annotations oRange [44]. Le choix de XML pour ce langage d'annotation est issu d'une volonté d'hériter de la nature extensible de ce format, et d'offrir une flexibilité dans la prise en compte des informations de flot fournies, à savoir la liberté d'intégrer l'ensemble des annotations ou seulement une partie dans l'analyse de WCET. Un extrait de la grammaire de ce langage est fournie en appendice A.

2.5 Conclusion

L'estimation de WCET par analyse statique repose conjointement sur des informations issues du matériel (les temps d'exécution de chaque instruction, les temps accès aux ressources, etc.) et sur l'intégration des diverses propriétés connues sur le programme en lui-même (les chemins d'exécution possibles, les bornes des boucles rencontrées, etc.). Les langages d'annotation sont une réponse efficace à l'expression et la prise en compte de ces diverses propriétés, mais ils rencontrent certaines limites que nous pointons dans cette conclusion. Dans le même temps, nous proposons d'utiliser des automates spécifiques en remplacement des langages d'annotation, et nous montrons comment ce choix pourrait permettre de surmonter les diverses limitations mises en évidence et offrir des perspectives d'amélioration de l'analyse de WCET.

2.5.1 Une base formelle

Les langages d'annotation ne sont pas toujours formellement spécifiés. Leur syntaxe et leur structure sont souvent supportées par une grammaire mais la sémantique qui les accompagne tend à se rapprocher du langage naturel pour simplifier l'expression et l'annotation par l'humain. De plus, le processus d'intégration lui-même n'est pas non plus formellement défini. Cela implique que pour intégrer une propriété sémantique dans un outil d'analyse de WCET, il faut soit que l'annotation ne souffre aucune ambiguïté (ce qui peut être difficile à assurer en l'absence de définitions formelles), soit se référer au producteur de l'annotation pour garantir que le résultat de l'intégration dans l'analyse ait le sens le plus proche possible de la propriété initiale. On voit ici que l'humain rentre beaucoup en compte alors que ces analyses sont rattachées aux systèmes critiques.

La solution proposée dans cette thèse repose sur la base formelle des automates

finis. Nous avons étendu et formalisé ces automates pour répondre aux problématiques de calcul de WCET et nous avons fourni un certain nombre d'opérations usuelles et spécifiques sur ces automates étendus. Des ambiguïtés persistent dans la traduction des annotations en automate mais ce sont les mêmes que lorsque un expert exprime des annotations à partir de propriétés sémantiques. Il est ainsi possible de simplement utiliser la propriété initiale pour définir ou confirmer l'exactitude de l'automate correspondant. D'autre part, il est possible pour un langage d'annotation donné, de mettre en place une génération automatique des automates correspondants qui se chargeront ensuite de l'intégration.

2.5.2 Une intégration contrôlée des annotations

L'article [33] met explicitement en avant le lien fort qui existe entre un langage d'annotation et la méthode d'estimation du WCET à laquelle il est associé : l'expressivité d'un langage d'annotation qui ne pourrait pas être exploitée par la méthode d'estimation est inutile, mais un langage dont l'expressivité est trop faible pour supporter des annotations pertinentes pour la méthode d'estimation de WCET l'est aussi. La phase d'intégration des annotation n'est cependant pas prise en considération ici, et nous pensons que pour la méthode d'estimation de WCET par IPET, l'intégration systématique des annotations de flot sous forme de contraintes ILP constitue deux sources de sur-estimation. La première concerne le fait que des contraintes locales à un contexte seront transformées en contraintes ILP globales, ce qui fera réapparaître certains des chemins infaisables initialement interdits par les contraintes locales. La seconde vient du fait que l'intégration d'une contrainte ILP n'aura pas d'impact sur les analyses bas-niveau par interprétation abstraite, alors que la structure même du CFG pourrait supporter certaines de ces contraintes par un dépliage partiel, ce qui pourrait réduire le pessimisme lié aux abstractions dans la modélisation des éléments matériels.

Par ailleurs, le suivi des variables tout au long de l'analyse de WCET ainsi que l'adaptation des contraintes demandent des efforts importants en cas d'évolution du CFG par exemple.

Pour toute annotation qu'il est possible de traduire en un automate enrichi que nous présentons dans cette thèse, l'intégration de la propriété dans l'analyse de WCET est assurée. De plus, la phase d'intégration offre un degré de liberté supplémentaire. D'une part, il est toujours possible d'intégrer dans l'analyse de WCET, les annotations issues

d'un automate par l'ajout d'une contrainte ILP, comme c'était le cas pour les langages d'annotation. Cependant cette forme d'intégration est gérée automatiquement et suit les modifications du CFG tout en adaptant les variables rattachées aux différentes contraintes. Il n'y a donc pas d'effort nécessaire pour le maintien de la cohérence générale entre les variables, les contraintes et le CFG. D'autre part, les automates d'annotation peuvent s'intégrer dans l'analyse de WCET par la modification structurelle du CFG, ce qui présente des avantages comme un potentiel d'amélioration dans la précision des analyses bas-niveau, et des inconvénients comme une possible explosion de la taille du CFG. Comme il est possible d'alterner et de combiner l'intégration sous forme structurelle et celle qui attrait au système ILP directement, notre proposition permet d'avoir un juste milieu entre la précision qu'il est possible de gagner par dépliage systématique du CFG et l'ajout de contraintes qui sera moins coûteux en terme de temps et taille d'analyse. Le coût du passage à l'échelle est ainsi entièrement contrôlable, et les limites rencontrées sont celles dont souffre habituellement la résolution du système ILP en lui-même.

2.5.3 Un outil évolutif

Lors de l'ajout de nouvelles classes d'annotations, il est nécessaire de travailler au niveau du langage en lui-même, pour exprimer correctement la propriété et définir la grammaire qui supportera cette nouvelle classe d'annotations. Il faudra cependant également modifier l'outil d'analyse de WCET, en utilisant la sémantique de ces annotations pour rendre cohérente leur intégration dans l'analyse, ou bien trouver comment générer les contraintes ILP correspondantes et maintenir leur cohérence tout au long de l'analyse.

Les automates que nous proposons permettent d'intégrer de manière automatique et systématique les annotations dans l'outil d'analyse. En amont, les automates d'annotations peuvent être générés à partir d'un langage d'annotation, auquel cas il faudra définir la génération de cet automate, mais rien n'interdit de définir la classe d'annotation directement comme un automate et transférer ainsi le travail qui se faisait au niveau du langage lui-même vers les automates.

2.5.4 Un support adapté aux annotations

Les langages d'annotation sont souvent associés à un seul type d'outil, ce qui limite leur portabilité et leur adaptabilité aux autres outils qui permettent de faire du WCET. Le langage FFX notamment a été développé dans ce sens et supporte cette idée de portabilité et d'harmonisation des langages. Cependant les différents outils d'analyse de WCET sont déjà liés à leurs propres langages d'annotation qui répondent à leur attentes spécifiques et l'adoption d'un langage commun demanderait du temps et des efforts de toutes parts. Le seul bénéfice qui en ressortirait serait l'harmonisation de ces langages, alors que peu d'outils ont comme objectif de travailler ensemble. Et d'autre part, il n'est pas exclu que le langage commun adopté souffre des autres limitations citées dans cette conclusion.

Les automates présentés dans cette thèse se veulent être une brique logicielle située soit à la place du langage d'annotation, soit entre ce langage et l'outil d'analyse de WCET. Ils sont donc décorrélés du langage en lui-même, mais il est possible de les dériver depuis un langage d'annotation particulier en définissant des règles de génération spécifique, ou ils peuvent être directement utilisés comme support primaire aux propriétés sémantiques. Adapter les différents outils à cette nouvelle brique n'est pas moins difficile que d'envisager un langage d'annotation commun, mais les réponses que les automates offrent aux problèmes de cette liste et les perspectives d'amélioration de la précision du WCET par une intégration plus efficace pourraient bénéficier à divers outils.

“Les chiffres, c’est pas une science exacte figurez-vous !”

— Karadoc de Vannes

3

Principe des automates d’annotation

Sommaire

3.1 Idée générale

L’ensemble des chemins réellement exécutables dans un programme est un sous-ensemble des chemins autorisés par son CFG. En effet, la représentation sous forme de CFG rassemble l’ensemble des exécutions structurellement possibles d’un programme. Or, par la suite, cet ensemble sert de base à l’analyse de WCET. Il convient donc

qu'il soit le plus proche possible de l'ensemble des chemins réels, et à cette fin, il est nécessaire d'apporter des informations qui vont préciser, parmi les chemins autorisés par le CFG, ceux qui ne sont pas réellement faisables.

On utilise pour cela des annotations de flot ou *flow facts*. Combiner le CFG avec une annotation qui précise une borne de boucle revient à supprimer des chemins auxquels on s'intéresse tous ceux qui itèrent un trop grand nombre de fois. On opère ici une intersection entre l'ensemble des exécutions qui correspondent aux chemins du CFG et l'ensemble des exécutions quelconques qui respectent la borne de boucle. Le résultat de cette intersection peut sembler complexe à spécifier mais sa forme est simplifiée si on passe par le complément, puisqu'on autorise en réalité tous les chemins sauf ceux qui ne respectent pas la borne de boucle.

Il est intéressant de remarquer plusieurs choses ici. Tout d'abord, l'intersection des ensembles permet de rendre le sur-ensemble de chemins autorisés par le CFG plus proche de l'ensemble des chemins réels. Par ailleurs, cette intersection existe également dans le domaine des automates puisque faire le produit de deux automates revient à faire l'intersection de leurs langages [27]. Puisque le CFG est défini comme un graphe orienté, il est possible de le représenter sous forme d'un automate à états fini étiqueté avec comme alphabet les paires de blocs ⟨source, destination⟩ des arcs du graphe. Ainsi, si on parvient à décrire une annotation de flot comme un automate à états fini, alors il suffira d'opérer un simple produit avec le CFG sous forme d'automate pour intégrer cette annotation dans la structure même de ce CFG.

3.2 Application à un exemple

Dans cette section, nous détaillons et nous illustrons par un exemple l'ensemble du processus permettant l'intégration d'annotations dans un CFG en utilisant des automates. Les étapes principales de cette méthode sont les suivantes :

- Transformation du CFG en un automate.
- Expression d'une annotation sous forme d'un automate.
- Produit entre les deux automates.
- Reconstruction du CFG à partir du résultat du produit.
- Analyse de WCET sur ce nouveau CFG qui intègre l'annotation dans sa structure.

3.2.1 Un CFG sous forme d'automate

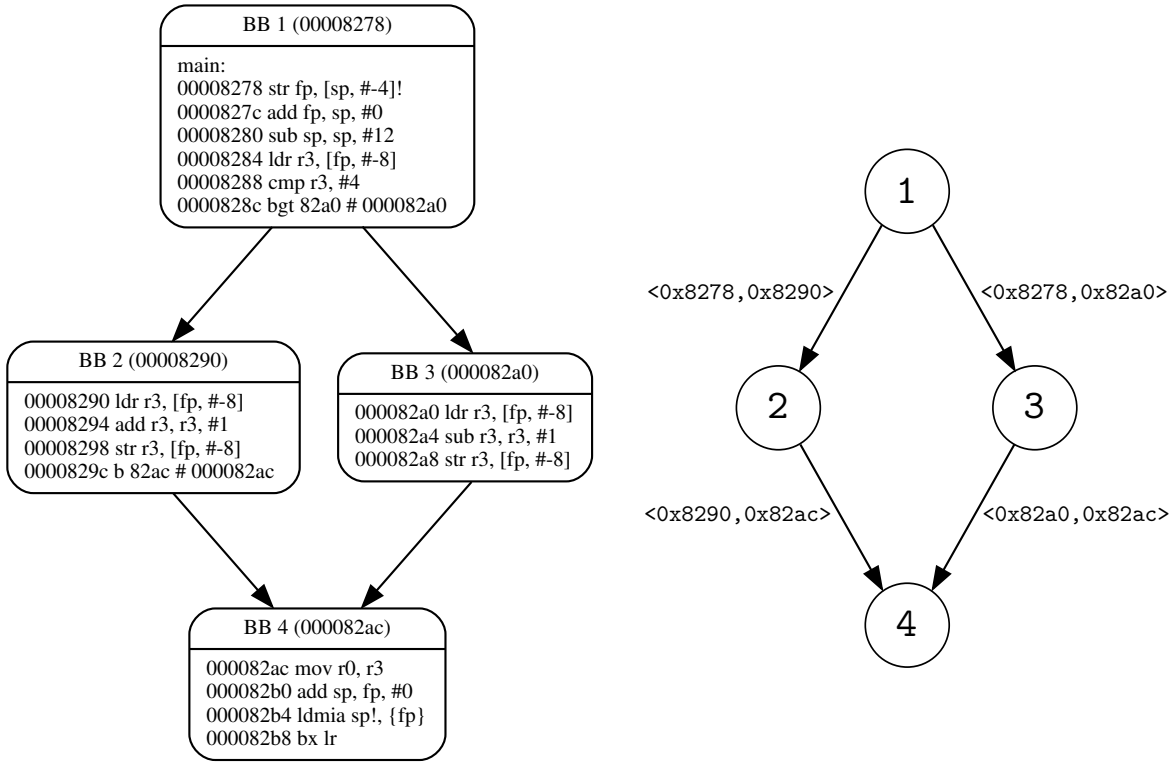


Figure 3.1 – Exemple de CFG (a) et l'automate correspondant (b).

Le CFG d'un programme est composé d'un ensemble de blocs de base et d'arcs qui les relient. Les blocs de base ont chacun une adresse physique correspondant à l'adresse de la zone mémoire qu'occupe leur première instruction. Les arcs quant à eux n'ont pas d'adresse à proprement parler mais ils relient les blocs en suivant les instructions en séquence ou les branchements qui composent le programme. On peut ainsi considérer qu'un arc est représenté par l'adresse de son bloc *source* et l'adresse de son bloc *destination*. On utilisera ce couple d'adresses pour fabriquer un automate correspondant au CFG.

On l'appellera automate de flot de contrôle ou *Control Flow Automaton* (CFA) pour le différencier du CFG dans la suite.

Dans la figure 3.1, le CFA (b) a comme alphabet des paires d'adresses issues des blocs du CFG (a). Notons que les noms des nœuds n'ont pas d'importance puisque l'information que l'on souhaite conserver, à savoir les paires d'adresses qui définissent une transition, est portée par les arcs de l'automate.

Les arcs comportent des paires d'adresses et non des paires de numéros de blocs de base car la numérotation d'un CFG n'est pas unique. Cependant lorsqu'on *inline* les appels de fonctions, c'est-à-dire qu'on inclut dans un même CFG les sous-graphes qui correspondent aux fonctions appelées, les adresses des blocs perdent leur unicité. Néanmoins le bloc de base qui correspond à une adresse est toujours le même et si on retrouve plusieurs blocs avec la même adresse, c'est en réalité le bloc entier qui a été dupliqué. Ainsi, pour reconstruire un CFG à partir du CFA correspondant, il suffit de retrouver n'importe quel bloc dont l'adresse est portée par les transition sortantes d'un état et de remplacer cet état par le bloc en question.

Notons que dans la suite de ce document, on utilisera des notations simplifiées pour certaines transitions : on nommera parfois explicitement un arc du CFG et on utilisera ce même nom sur un automate, sans donner le couple d'adresses ⟨source, destination⟩ correspondantes, dans le but de faciliter la compréhension.

3.2.2 Une annotation sous forme d'automate

Considérons qu'un expert ou qu'une analyse automatique soit capable de détecter que la branche droite du CFG de la figure 3.1a correspond à du code mort [1] et que la branche gauche est donc toujours exécutée. Il semble pertinent d'intégrer cette annotation dans l'analyse de WCET. Habituellement, cette annotation sera décrite dans un langage d'annotation et mise à disposition de l'analyseur. Pour des cas simples comme celui-ci, certains analyseurs performants sont capables de transformer leur CFG à la volée, auquel cas ils supprimeront le bloc de base 3 et les arcs rattachés. Cependant, dans le cas contraire, c'est dans le système ILP que l'annotation sera intégrée avec une nouvelle contrainte du type $e_{1-3} = 0$ qui exprimera que l'arc entre les blocs 1 et 3 doit être empruntée zéro fois. Notons que l'alternative qui consiste à utiliser la contrainte $e_{1-2} = 1$ est équivalente dans le cas qui nous intéresse mais pose plusieurs problèmes : tout d'abord lorsqu'une branche du CFG est annotée comme contenant du code mort, il n'y a aucune information sur les autres branches. Pour déduire cette contrainte, il faudrait donc faire une analyse du CFG alors que ce n'est pas nécessaire avec la première contrainte. De plus, si ici le fait d'interdire la branche de droite force le passage dans la branche de gauche, ça ne sera pas le cas si il y a plus d'une branche alternative. Enfin, cette seconde contrainte est plus difficilement généralisable si elle apparaît dans une

1. code qui n'est jamais exécuté

boucle par exemple, alors que la première contrainte restera inchangée. Pour toutes ces raisons on privilégie en règle générale l'expression et l'intégration d'annotations qui interdisent un chemin spécifique qu'on appelle alors *chemin infaisable*.

Les mêmes alternatives s'offrent à nous si on souhaite définir un automate qui supporterait cette annotation, à savoir que celui-ci peut soit autoriser uniquement la branche de gauche, soit interdire uniquement la branche de droite. Cependant l'annotation doit pouvoir être traduite en automate sans passer par une analyse du CFG, ce qui implique que les données de l'annotation doivent être suffisantes pour construire l'automate correspondant. Cette raison, à laquelle s'ajoutent celles citées ci-dessus, nous pousse à choisir une définition d'automates qui interdit la branche spécifiée par l'annotation.

Ce type d'automate a pour rôle d'exprimer des annotations de flots, ou *flow facts*. On les appellera donc automates d'annotation de flot ou *Flow Fact Automata* (FFAs) dans la suite de ce document.

La figure 3.2 présente un FFA qui autorise tous les chemins sauf ceux qui empruntent un arc entre le bloc d'adresse `0x8278` et le bloc d'adresse `0x82a0`.

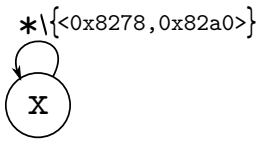


Figure 3.2 – Exemple de FFA qui interdit un arc spécifique du CFG.

On peut noter que l'automate est compact en terme de nombre d'états, et qu'une transition réflexive est suffisante pour intégrer l'annotation. Il est déterministe, mais pas complet, puisqu'on pourrait rajouter un état "poubelle" pour la transition étiquetée ⟨`0x8278`,`0x82a0`⟩. On rajouterait cependant un état, qui serait ensuite susceptible de dupliquer les états du CFA lors du produit et donc d'augmenter la complexité du résultat. À l'inverse, le produit d'un CFA avec un automate à un seul état n'ajoutera pas d'états par construction.

3.2.3 Un produit entre un CFA et un FFA

Le résultat du produit entre le CFA de la figure 3.1b et le FFA de la figure 3.2 est présenté sur la figure 3.3a. On observe que la branche de droite du CFA a bien disparu puisque la transition étiquetée ⟨`0x8278`,`0x82a0`⟩ est interdite dans le FFA.

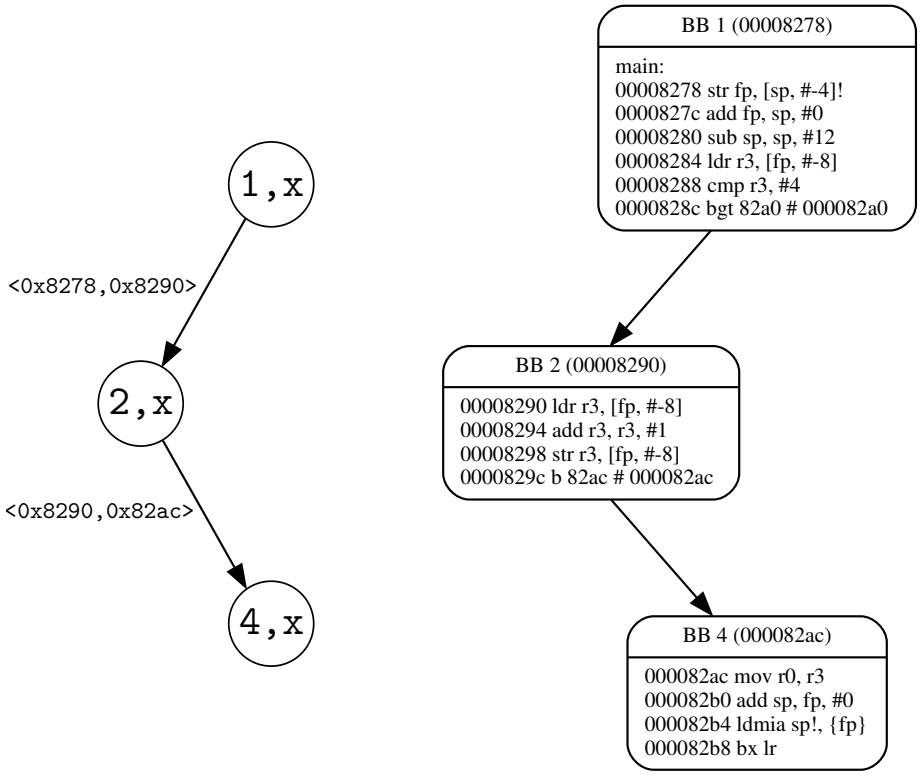


Figure 3.3 – Résultat de FFA × CFA (a) et le CFG correspondant (b).

3.2.4 Un nouveau CFG

La correspondance entre le résultat du produit présenté en figure 3.3a et le CFG reconstruit de la figure 3.3b est aisée à faire. On peut noter que les numéros `1`, `2` et `4` des états de l'automate sont bien les numéros des blocs de base du CFG, mais surtout que les labels des transitions de l'automate correspondent bien. Pour les raisons expliquées dans la partie 3.2.1, on se basera sur les labels des transitions de l'automate pour retrouver les blocs de base qui nous intéressent et les arcs du CFG correspondants. On notera donc que l'annotation de code mort a effectivement été intégrée dans la structure même du CFG comme si l'analyseur avait été capable d'en couper des branches à la volée et sans pour autant avoir eu besoin d'enrichir le système ILP avec une nouvelle contrainte.

3.3 Le cas critique des bornes de boucles

Les FFAs visent le support des annotations de flot qui sont utilisées dans le calcul du WCET. Or, les bornes de boucles sont une information sémantique primordiale puisque sans elles, il est impossible de calculer un WCET qui ne soit pas infini. Il est donc nécessaire de trouver un moyen de supporter l'expression et l'intégration de ces bornes via les FFAs.

On appelle borne de boucle une limite supérieure au nombre d'itérations de cette boucle. Il existe évidemment un nombre infini de bornes pour une même boucle qui diffèrent simplement par leur précision. On parlera dans le cas général de boucle bornée à partir du moment où on a réussi à définir une borne finie. Dans la représentation sous forme de graphe d'un programme (CFG), on trouve des cycles qui correspondent à ces boucles. C'est donc pour les arcs qui composent ces cycles qu'il advient de préciser un nombre maximal d'apparitions dans les exécutions du programme.

Dans le CFG, lorsque l'on souhaite préciser un nombre d'itérations maximal d'une boucle, on borne généralement son arc retour (*backedge*), ce qui limite naturellement le nombre de retour à la tête de boucle. Dans l'analyse de WCET par la méthode IPET, on associe une variable ($x_{\texttt{backedge}}$) à cet arc retour, et on propage l'information de borne de boucle sous forme de contrainte ILP relative à cette variable. Dans le cas d'une boucle simple[2], cette contrainte s'écrit : $x_{\texttt{backedge}} \leq n$, avec n la borne de boucle.

Par la suite, l'utilisation de la méthode IPET et sa résolution par ILP permettra de propager cette contrainte par la loi des nœuds à l'ensemble des blocs et des arcs qui composent la boucle comme précisé dans la partie 2.3.4.3.

Exprimer une borne de boucle avec les FFAs actuels

La nature des FFAs tels qu'ils sont définis actuellement permet la construction d'un automate capable de supporter une restriction sur le nombre d'itérations d'une boucle.

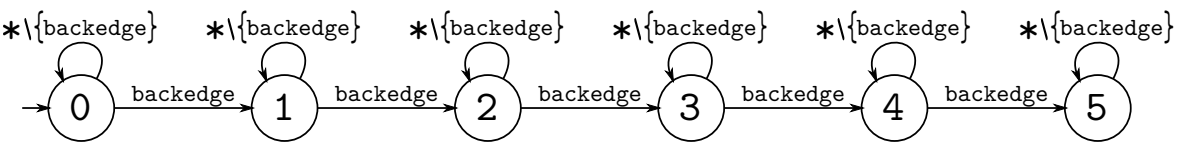


Figure 3.4 – Exemple de FFA exprimant une annotation équivalent à $x_{\texttt{backedge}} \leq 5$

2. sans imbrication de boucles

La figure 3.4 illustre un FFA capable de supporter une information de borne de boucle. L'expression d'une information numérique passe ici par l'augmentation du nombre d'états de cet automate, ce qui permettra, après un produit avec le CFG de complètement déplier la boucle autant de fois que la borne l'autorise. Le résultat de ce dépliage correspond bien à ce qui était recherché, à savoir restreindre structurellement les chemins possibles du CFG en fonction de la borne de boucle. Néanmoins, ce dépliage complet pose des problèmes de passage à l'échelle et va à l'encontre de la représentation compacte offerte par le CFG et qui sert de base à l'analyse statique de WCET.

Enrichir les FFAs apparaît ainsi comme une nécessité afin d'obtenir un formalisme plus puissant et plus expressif capables de supporter une représentation plus compacte de propriétés numériques qui ne soit pas basée sur les seuls états de ces automates.

3.3.1 La piste des automates à compteurs

Le formalisme des automates à compteurs [67, 13] vise à représenter des comptes et des décomptes en faisant évoluer des variables appelées *compteurs* (par incrément, décrément, additions de compteurs et de constantes, etc.) et en "gardant" des transitions, c'est-à-dire en n'autorisant une transition que lorsque les contraintes qui y sont rattachées sont respectées. Les compteurs utilisés doivent naturellement être initialisés, soit lors d'une phase d'initialisation, soit en précisant un état initial des variables de l'automate.

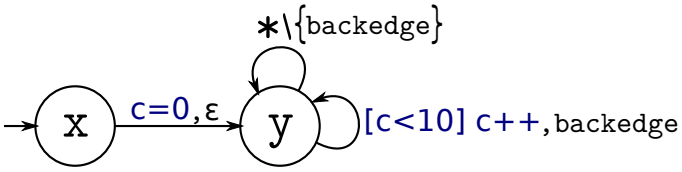


Figure 3.5 – Exemple d'automate à compteurs pour borner une boucle.

L'automate à compteur de la figure 3.5 présente un automate à compteurs permettant de borner une boucle en autorisant à prendre son arc retour dix fois au maximum.

On peut remarquer tout d'abord une phase d'initialisation pour la variable `c`, avec une transition étiquetée par ε, aussi appelé chaîne vide. La transition qui supporte ce caractère est une ε-transition, c'est-à-dire une transition spontanée qui ne consomme pas de symbole d'entrée [27]. On l'utilise ici spécifiquement pour l'initialisation des

variables mais on pourrait tout aussi bien préciser que l'état initial des variables de l'automate est {c=0}.

Lorsque l'automate est dans l'état (y), la transition étiquetée backedge est gardée par la condition [c<10]. En d'autres termes, il ne sera possible d'effectuer cette transition que tant que la contrainte sera vérifiée. On note également que c'est cette même transition qui incrémente la variable c. Quand sa valeur atteindra 10, l'arc retour deviendra une transition interdite et on bornera ainsi la boucle.

Dans l'article [42], Metzner modélise les durées d'exécution des blocs du CFG par des automates similaires aux automates à compteurs, dans une méthode d'estimation de WCET par *model checking*. D'autres travaux qui suivent la même approche [15, 5, 10] pour estimer le WCET utilisent généralement le *model checker* UPPAAL [6] et modélisent l'architecture matérielle par des *timed automata* [3]. Dans ce type d'automate, on trouve des compteurs qui évoluent avec le temps, et autorisent ainsi des transitions gardées par des contraintes de durées. Dans cette approche par *model checking*, les automates à compteurs sont utilisés pour décrire, contraindre, et surveiller des systèmes dans lesquels des paramètres évoluent régulièrement.

Notre objectif est différent puisque l'approche que nous visons est l'estimation de WCET par la méthode IPET. Cette méthode ne requiert pas de suivre l'évolution des différents compteurs puisque seules leurs valeurs finales sont pertinentes. Ces différents compteurs seront à terme intégrés dans le système ILP décrivant les contraintes structurelles du CFG associées à l'ensemble des contraintes de flot issues des annotations, et c'est la résolution de ce système ILP qui permettra d'obtenir le WCET du programme.

Dans [33], Kirner et al. mettent en avant le fait que l'expressivité d'un langage d'annotation doit être adaptée à la méthode d'estimation du WCET visée. Ainsi, si les automates à compteurs pourraient permettre de véhiculer les annotations de flot à destination de la méthode IPET, il est évident que leur expressivité est bien supérieure à ce que la méthode en elle-même est capable de supporter – on parlera dans ce cas d'*overkill* – et qu'il est possible d'utiliser un modèle plus simple. D'autre part, la méthode IPET consiste à définir des variables pour les éléments du CFG et à exprimer un ensemble de contraintes sur ces variables. L'idée d'utiliser ces mêmes types de variables associées à des contraintes globales permettrait d'avoir une expressivité adaptée à l'approche et faciliterait dans le même temps le transfert des variables et contraintes des automates vers des variables et des contraintes ILP.

3.3.2 Exemple d'automate à contraintes globales

Dans l'analyse de WCET, les contraintes ILP associées au CFG sont relatives à des arcs/blocs du CFG. Afin de supporter ce type de contrainte dans les FFAs, nous avons décidé d'attacher les contraintes aux automates et de placer les variables desquelles elles dépendent sur les transitions de l'automate.

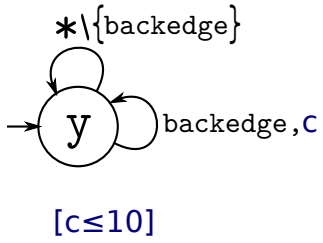


Figure 3.6 – Exemple d'automate à contraintes globales.

La figure 3.6 montre un automate accompagné d'une contrainte globale équivalent à l'automate à compteurs de la figure 3.5. Les transitions ne sont pas gardées et elles ne font pas non plus évoluer les compteurs par incrément, affectation, etc. Les variables (e.g. "`c`") sont simplement attachées aux transitions (e.g. "`backedge`") et utilisées dans les contraintes globales (e.g. "`[c≤10]`"). Le fonctionnement de cet automate est lié au langage qu'il reconnaît. Ainsi, un mot est accepté par cet automate si :

1. il existe un ensemble d'états de cet automate capable de consommer dans l'ordre toutes les lettres de ce mot en remplissant une liste composée des occurrences des variables rencontrées,
2. cette liste d'occurrences des variables satisfait les contraintes de l'automate.

On notera que l'automate ne compte pas directement le nombre de fois que la transition `backedge` est empruntée. C'est seulement l'utilisation de la contrainte associée qui permet de réduire les traces acceptées par cet automate. C'est donc l'ensemble des transitions empruntées lors de la consommation d'un mot qui doit permettre de respecter les contraintes de l'automate.

D'autre part, dans l'analyse de WCET, on n'utilisera pas ces automates pour vérifier directement l'acceptation d'un mot, mais pour associer les variable aux arcs du CFG, et intégrer les contraintes dans le système ILP utilisé dans IPET.

3.4 Définitions formelles

Cette partie décrit mathématiquement les automates à contraintes, les fonctions de transitions et le langage accepté par ces automates. Ces automates sont une extension des automates finis déterministes ou *Deterministic Finite Automata* (DFAs) et quand ça sera possible, nous utiliserons les notions et notations déjà définies dans la littérature.

Notation : On utilisera notamment les notations $\mathcal{X}^\star$ ou $\Sigma^\star$, c'est-à-dire l'étoile [3] de Kleene [35] appliquée à l'ensemble des variables $\mathcal{X}$ ou à l'alphabet Σ, ce qui inclus le mot vide qu'on notera ε.

3.4.1 Automates à contraintes globales

Définition 4. *Un automate à contraintes globales est un tuple* $(N, i, \Sigma, \mathcal{X}, \Phi, \delta)$ *t.q. :*

- N *est un ensemble fini d'états,*
- $i \in N$ *est l'état initial de l'automate,*
- Σ *est un alphabet fini,*
- $\mathcal{X}$ *est un ensemble de variables,*
- $\Phi : (\mathcal{X} \to \mathbb{N}) \to \text{bool}$ *est une conjonction de contraintes linéaires sur* $\mathcal{X}$*,*
- $\delta : N \times \Sigma \to (N \cup \{\bot\}) \times \mathcal{X}^\star$ *est une fonction de transition entre les états qui consomme une lettre de l'alphabet et renvoie l'état d'arrivée de la transition ou* $\bot$ *s'il n'en existe pas, ainsi qu'une liste de variables*

L'alphabet Σ : Nous souhaitons utiliser des automates pour exprimer des informations sur les chemins du CFG, et donc sur les blocs de base. Pour cela, il est possible d'utiliser soit sur leurs numéros, soit leurs adresses physiques. Les annotations sont par nature décrites séparément du programme, ce qui implique d'utiliser des informations immuables. Or la numérotation des blocs de base d'un CFG n'est pas unique, même si elle a l'avantage d'associer un unique bloc de base à chaque numéro. À l'inverse, les adresses physiques des blocs seront les mêmes quelle que soit la représentation, mais plusieurs blocs peuvent avoir la même adresse physique, notamment lorsqu'on travaille sur un CFG *inliné*. Néanmoins, l'absence d'unicité de la numérotation nous oblige à choisir comme alphabet des paires d'adresses physiques $< \texttt{adr}_x, \texttt{adr}_y >$ qui

3. On différenciera l'étoile de Kleene "$\star$" du caractère "$*$" utilisé pour les transitions par défaut.

décriront les arcs du CFG existants entre les blocs d'adresses $\texttt{adr}_x$ et $\texttt{adr}_y$. Une paire d'adresses pourra donc correspondre à plusieurs arcs du CFG, ce qui représente à la fois un avantage car on peut répercuter une annotation sur tous les arcs correspondantes dans le CFG, et un inconvénient car il est nécessaire d'utiliser le contexte d'appel pour discriminer deux paires d'adresses. Ce point fera l'objet du chapitre 4.

D'autre part, dans l'analyse de WCET, on s'intéresse souvent à une ou quelques transitions spécifiques du CFG (celles qu'on souhaite interdire), sans se soucier (ni même connaître) toutes les autres transitions existantes. Pour cette raison, une représentation compacte d'ensemble de transitions est nécessaire. Nous utiliserons les notations habituelles de la littérature pour exclure certaines transition spécifiques de l'ensemble Σ. Ainsi, nous écrirons $\Sigma \backslash \{< \texttt{adr}_x, \texttt{adr}_y >\}$ ou encore $* \backslash \{< \texttt{adr}_x, \texttt{adr}_y >\}$ pour exprimer l'ensemble des transitions existantes dans le CFG (l'ensemble des paires de l'alphabet) sauf les transitions $\texttt{adr}_x \rightarrow \texttt{adr}_y$.

La fonction δ : Une particularité de ces automates est la possibilité de renvoyer une liste de variables lors de la transition entre les états. Considérons deux états $n_1, n_2 \in N$, une lettre $a \in \Sigma$ et une variable $\gamma \in \mathcal{X}$. S'il existe une transition de l'état n_1 à l'état n_2 qui consomme la lettre a et à laquelle est associée la variable γ, alors le résultat de la fonction de transition appliquée à n_1 et a s'écrira :

$$\delta(n_1, a) = \langle n_2, \gamma \rangle$$

La paire renvoyée par la fonction de transition est donc bien composée de l'état d'arrivée n_2 et d'une liste de variables (ici composée d'un seul élément : γ).

Une autre différence notable avec les DFA est l'absence d'états terminaux. En effet, nos automates ne sont pas complets, afin de réduire leur nombre d'états. La fonction de transition δ peut donc renvoyer $\perp$ s'il n'existe pas de transition capable de consommer la lettre demandée depuis l'état courant afin de signifier une erreur. $\perp$ n'est pas un *état* à proprement parler mais on considérera que $\forall a \in \Sigma$, $\delta(\perp, a) = \langle \perp, \varepsilon \rangle$, de façon à propager cette erreur si on essaye de consommer n'importe quel lettre de l'alphabet.

3.4.2 Extension de la fonction de transition aux mots

Cette section présente l'extension aux mots de la fonction de transition δ définie sur les lettres dans les automates à contraintes (partie 3.4.1). Cette extension est adaptée de celle définie sur les DFA [27], afin de l'appliquer correctement aux automates à contraintes, qui ont la particularité de propager une liste de variables. On précisera également la propagation d'une erreur rencontrée par la fonction de transition δ. On notera $\hat{\delta}$ cette fonction de transition étendue aux mots.

Définition 5. *Soient un automate à contraintes $\mathcal{A} = (N, i, \Sigma, \mathcal{X}, \Phi, \delta)$, deux états $n, n' \in N$, un mot $\omega \in \Sigma^\star$, une lettre $a \in \Sigma$ et deux listes de variables $L, X \in \mathcal{X}^\star$. On définit la fonction de transition $\hat{\delta} : N \times \mathcal{X}^\star \times \Sigma^\star \to N \times \mathcal{X}^\star$ à partir de l'état n ainsi :*

$$\begin{cases} \hat{\delta}(n, L, a\omega) = \hat{\delta}(n', L \cdot X, \omega) \text{ avec } \delta(n, a) = \langle n', X \rangle \\ \hat{\delta}(n, L, \varepsilon) = \langle n, L \rangle \end{cases}$$

Cette définition consomme un mot de gauche à droite en utilisant la fonction de transition δ qui renvoie une paire $\langle$*état*, *liste de variables*$\rangle$ sur chaque lettre jusqu'à trouver le mot vide. $\hat{\delta}$ peut également renvoyer $\perp$ à la place de n si un des appels à la fonction δ renvoie $\perp$ à la place de l'état, qui, comme expliqué dans la partie 3.4.1, sera ainsi propagé jusqu'au retour global de la fonction $\hat{\delta}$.

3.4.3 Validation de contraintes par une liste de variables

Les contraintes globales doivent être vérifiées pour valider l'acceptation d'un mot. Il est donc nécessaire pour cela de définir une fonction de validation d'un jeu de contraintes par une liste de variables.

Notation : On notera $|L|_\alpha$ le nombre d'occurrences de α dans la liste L.

Définition 6. *Soient $\mathcal{X}$ un ensemble de variables, $L \in \mathcal{X}^\star$ une liste de variables et Φ une conjonction de contraintes linéaires sur $\mathcal{X}$. On définit la fonction qui accepte L comme solution de Φ ainsi :*

$$\mathit{valid}(L, \Phi) = \Phi(\lambda x.|L|_x)$$

Cette formule opère une substitution des variables de l'ensemble $\mathcal{X}$ par leur nombre

d'occurrence dans la liste L, et vérifie si la conjonction de contraintes Φ est vraie lorsqu'on l'applique à ce nouvel ensemble de valeurs. En d'autres termes, la fonction *valid* retourne *vrai* si la liste L représente une solution de Φ.

3.4.4 Langage accepté par un automate à contraintes

Le langage accepté par un automate correspond à l'ensemble des mots qu'il reconnaît. On peut se baser sur la définition étendue de la fonction de transition aux mots de la partie 3.4.2 et sur la fonction qui vérifie qu'une liste de variables valide un jeu de contrainte de la partie 3.4.3 pour définir le langage accepté par un automate à contraintes.

Définition 7. *Soit $\mathcal{A}$ un automate à contraintes $(N, i, \Sigma, \mathcal{X}, \Phi, \delta)$. le langage accepté par cet automate noté $\mathcal{L}(\mathcal{A})$ est défini ainsi :*

$$\mathcal{L}(\mathcal{A}) = \left\{ \omega \in \Sigma^\star \mid \hat{\delta}(i, \varepsilon, \omega) = \langle n, L \rangle \wedge \mathit{valid}(L, \Phi) \text{ avec } n \in N \text{ et } L \in \mathcal{X}^\star \right\}$$

En d'autres termes, un mot peut être rejeté pour deux raisons :

- si une lettre d'un mot n'a pas pu être consommée, c'est-à-dire que la fonction $\hat{\delta}$ a renvoyé $\perp \notin N$.
- si la liste de variables retournée ne satisfait pas le jeu de contraintes Φ, c'est-à-dire si *valid* a renvoyé *faux*.

3.4.5 Produit d'automates à contraintes

Le produit entre le CFA et un FFA présenté dans la partie 3.2.3 permet l'intégration des annotations supportées par le FFA dans l'analyse de WCET. Ce produit est un cas particulier dans le sens ou le CFA est simplement une transformation du CFG et que par construction, il ne comporte pas de contraintes ni de variables. On peut donc aisément simuler un CFA avec un FFA qui aurait un ensemble de variables vide et une conjonction de contraintes vide. Pour ces raisons, nous ne définissons pas le simple produit CFA $\times$ FFA mais plutôt le cas général du produit de deux automates à contraintes, ce qui permettra par ailleurs de faire des produits de FFAs porteurs d'annotations différentes.

Les automates à contraintes sont une extension des DFAs. Par conséquent, certaines

définitions seront les mêmes pour le produit, notamment pour l'alphabet et les états, mais des précisions sont nécessaires concernant les variables, les contraintes, ainsi que la fonction de transition.

Définition 8. *Soient deux automates à contrainte* $\mathcal{A}_1 = (N_1, i_1, \Sigma, \mathcal{X}_1, \Phi_1, \delta_1)$ *et* $\mathcal{A}_2 = (N_2, i_2, \Sigma, \mathcal{X}_2, \Phi_2, \delta_2)$. *On notera* $\times$: *FFA* $\times$ *FFA* $\to$ *FFA le produit de ces automates défini ainsi :*

$$\mathcal{A}_1 \times \mathcal{A}_2 = \big(N_1 \times N_2, (i_1, i_2), \Sigma, \mathcal{X}_1 \uplus \mathcal{X}_2, \Phi_1 \wedge \Phi_2, \delta\big)$$

avec $\forall a \in \Sigma, n_i \in N_i, L_i \in \mathcal{X}_i^\star$ *:*

$$\delta\big((n_1, n_2), a\big) = \begin{cases} \big\langle (n_1', n_2'), L_1 \cdot L_2 \big\rangle \text{ pour } \begin{cases} \delta_1(n_1, a) = \langle n_1', L_1 \rangle \\ \delta_2(n_2, a) = \langle n_2', L_2 \rangle \end{cases} \\ \langle \bot, \varepsilon \rangle \text{ si } \delta_1(n_1, a) = \langle \bot, \varepsilon \rangle \vee \delta_2(n_2, a) = \langle \bot, \varepsilon \rangle \end{cases}$$

L'alphabet Σ : Nous souhaitons par cette opération effectuer l'intersection des langages. Il est donc logique de travailler sur les mêmes mots et donc de définir ce produit sur un alphabet commun Σ. En outre, on retrouve par construction les couples d'états ainsi que le couple d'états initiaux qui existent aussi dans le produit de DFAs.

L'ensemble de variables $\mathcal{X}_1 \uplus \mathcal{X}_2$: Le langage accepté par chaque automate à contraintes dépend du respect des conjonctions de contraintes (Φ) définies sur leur ensemble de variables ($\mathcal{X}$). Le langage accepté par le produit des deux automates dépend donc des deux ensembles de contraintes (Φ_1 et Φ_2) qui doivent être vérifiés avant l'acceptation d'un mot. Il convient donc de faire la conjonction de ces deux jeux de contraintes. Cependant, il est essentiel que les variables soient différentes d'un automate à l'autre, ce qui implique un renommage si nécessaire. En effet, l'intégrité des contraintes serait menacée si des variables du premier automate apparaissaient dans les contraintes de l'autre, et inversement. C'est la raison pour laquelle nous faisons l'union disjoint $\uplus$ des ensembles de variables $\mathcal{X}_1$ et $\mathcal{X}_2$.

La fonction δ : La fonction de transition δ permet, en consommant une lettre de l'alphabet, de passer d'un couple d'état à un autre et peut supporter une liste de variables $L \in \mathcal{X}^\star$. Le couple d'états dépend des résultats des transitions lorsque les automates initiaux consomment cette même lettre, et la liste de variables est concaténée

avec les variables résultantes de ces mêmes transitions. Notons que si un des deux automates ne peut pas consommer cette lettre (si sa fonction de transition renvoie $\perp$), alors la fonction de transition du produit renverra également $\perp$. Ainsi, seules les transitions communes aux deux automates seront présentes dans l'automate produit.

3.5 Illustration du produit

Le résultat du produit formellement défini dans la partie 3.4.5 est lui aussi un automate à contraintes globales dans lequel on retrouve l'union disjointe des variables et l'ensemble des contraintes présentes sur les deux automates initiaux.

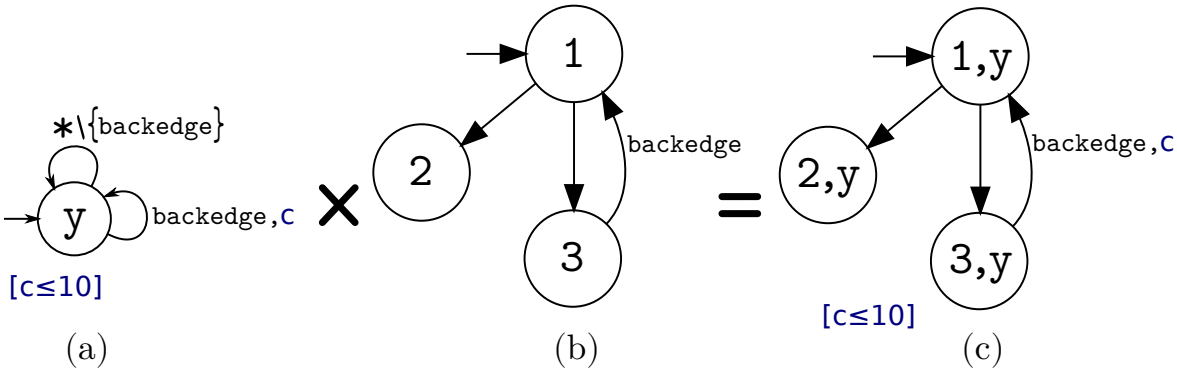


Figure 3.7 – Exemple de produit d'un FFA à contraintes globales (a) avec un CFA (b). Le résultat (c) est aussi un automate à contraintes globales.

La figure 3.7 illustre la méthode du produit entre un CFA et un FFA qui permet l'intégration d'une annotation de borne de boucle dans un CFG. Le CFA (b) comporte une boucle entre les états 1 et 3 et le FFA (a) supporte une contrainte [c≤10] qui, associée à la variable `c` attachée à la transition `backedge`, restreint les mots acceptés à ceux qui n'empruntent pas plus de dix fois cet arc.

Lors de l'opération de produit, toutes les transitions du CFA autres que l'arc retour sont acceptées par le FFA sur la transition étoile (*) ce qui permet d'avoir la même structure que le CFA dans le résultat. La transition `backedge` quant à elle est enrichie de la variable `c` présente sur la transition du FFA correspondante, et la contrainte globale est également rattachée au résultat de la figure 3.7c.

La figure 3.8 illustre un produit avec le même FFA que la figure 3.7, mais avec deux transitions `backedges` dans le CFA. Nous montrerons dans le chapitre 5 qu'il est

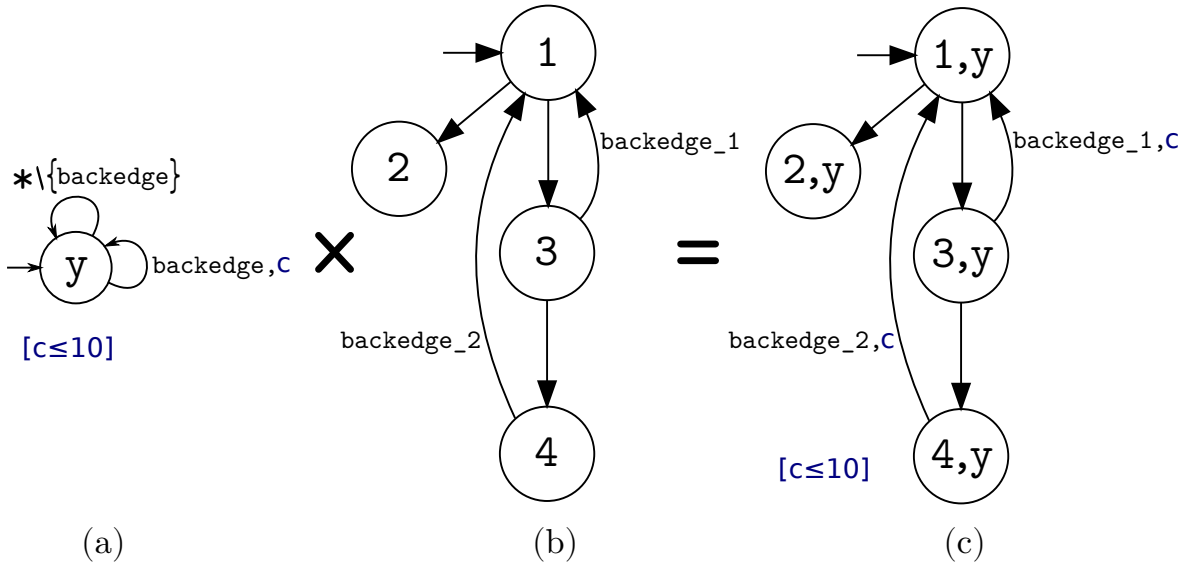


Figure 3.8 – Exemple de produit d'un FFA à contraintes globales (a) avec un CFA (b). La variable `c` est répliquée sur les `backedges` du résultat (c).

possible de décrire un ensemble d'arcs du CFG avec une seule transition d'un FFA [4] et donc que le produit mettra bien en correspondance la transition `backedge` du FFA avec chacun des deux arcs retours du CFA, tout en y attachant la variable `c`. On retrouve également la transition réflexive étoile (*) sur l'état (y) qui permet de maintenir la structure du CFA, ainsi que le transfert telle quelle de la contrainte globale.

On observe que la distribution de la variable `c` maintient la cohérence de l'annotation, puisqu'on interdit bien les traces qui empruntent plus de dix fois une transition `backedge` quelconque. Remarquons qu'on peut aisément envisager de séparer la variable `c` en deux variables c_1 et c_2 et de changer la contrainte en $[c_1+c_2 \leq 10]$. Néanmoins, cette séparation est faite systématiquement lors de la phase de reconstruction du CFG à partir du CFA (partie 3.5.1), et ce afin de différencier les arcs dans le système ILP. Il n'est donc pas nécessaire de faire cette séparation pour le moment, d'autant que si le CFA vient à être modifié par la suite, il faudra faire attention à maintenir la cohérence des variables. C'est pourquoi on choisit de reporter cette séparation à la phase de reconstruction du CFG.

4. On décrira par exemple une transition dont la sémantique serait : "tous les arcs qui reviennent à la tête de boucle"

3.5.1 Reconstruction du CFG et adaptation des contraintes

Nous avons montré dans la partie 3.2.4 qu'il était possible de reconstruire un CFG avec le résultat d'un produit CFA × FFA. Il suffit pour cela de retrouver les blocs de base à partir des labels des transitions de ce résultat. Cependant, après l'enrichissement du formalisme avec des variables et des contraintes, il convient de préciser comment ces éléments sont transférés au CFG et au système ILP.

Lors de l'application de IPET, on transforme les contraintes structurelles du CFG en contraintes linéaires sur des variables qui correspondent aux différents éléments du graphe. En particulier, la méthode génère une variable pour chaque arc du graphe, qui sont ainsi très proches des variables que nous *accrochons* aux transitions des automates d'annotation. Il serait donc souhaitable d'exprimer les contraintes portées par le FFA en utilisant les variables ILP existantes afin de les mettre à disposition de l'analyseur statique. Cependant, dans le système ILP, les arcs du graphes sont distinguées de façon unique par les numéros des blocs de base, alors que les transitions qu'on trouve dans les automates d'annotation dépendent des adresses physiques de ces mêmes blocs. Il est donc nécessaire d'établir une correspondance entre les variables des FFAs attachées aux adresses des blocs et les variables ILP attachées aux numéros des blocs.

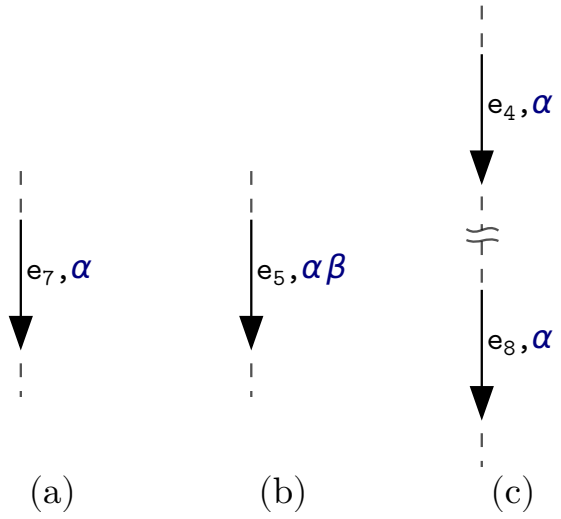


Figure 3.9 – Vue simplifiée de FFAs dans lesquels sont illustrés les différents cas de correspondance entre les variables des automates et les variables ILP.

Les cas de figure qui peuvent être rencontrés dans un FFA concernant la correspondance entre les variables portées par les FFAs (α, β) et les variables ILP (e_i) sont illustrés dans la figure 3.9. Nous fournissons ci-dessous les règles de correspondance à appliquer pour transférer les contraintes des automates vers des contraintes ILP :

3.9a. Une seule variable ILP correspond à une seule variable du FFA. Ce cas est le plus trivial puisqu'il suffit d'ajouter au système ILP les contraintes du FFA en remplaçant chaque occurrence de la variable du FFA par la variable ILP correspondante (par exemple, la contrainte $[\alpha \leq 8]$ deviendrait $[\mathtt{e}_7 \leq 8]$).

3.9b. Une seule variable ILP correspond à plusieurs variables du FFA. Ici encore, afin de conserver la cohérence des contraintes transférées depuis le FFA vers le système ILP, il suffira de remplacer chaque occurrence des variables de l'automate par la variable ILP associée (e.g. la contrainte $[\alpha + 2\beta \leq 8]$ deviendrait $[\mathtt{e}_5 + 2\mathtt{e}_5 \leq 8]$).

3.9c. Une même variable du FFA correspond à deux variables ILP. Lors du transfert des contraintes du FFA impliquant cette variable FFA vers une contrainte ILP équivalente, il faudra ici prendre soin de remplacer cette variable par la somme des variables ILP associées (e.g. la contrainte $[3\alpha \leq 7]$ deviendrait $[3(\mathtt{e}_4 + \mathtt{e}_8) \leq 7]$).

Le cas 3.9c correspondra généralement à un produit entre un FFA dans lequel on aurait attachée une variable α sur un arc identifié par des adresses physiques et un CFA dans lequel ce même arc se retrouverait dupliqué a deux endroits différents (conséquence de l'analyse de CFG *inlinés*). On aurait dans ce cas une variable ILP différente pour chaque apparition de cet arc dans le CFA, mais la variable du FFA serait distribuée sur ces deux arcs.

3.5.2 La gestion des contextes

Le formalisme présenté dans ce chapitre supporte la notion de contexte d'appel pour des cas suffisamment simples. Nous présentons dans cette section un exemple qui montre les bénéfices de l'utilisation d'informations de flots contextuelles. Nous illustrons ensuite comment le formalisme actuel est capable de supporter ce type d'annotations et nous en pointons les limitations qui seront levées au chapitre 4.

3.5.2.1 Exemple avec deux appels de fonction

Le programme présenté dans le code 3.1 comporte deux appels à la même fonction dans laquelle on trouve une boucle `for`. La boucle itère quarante fois lors du premier appel à cette fonction `f`, et soixante fois lors du second. En l'absence de contexte pour les annotations, la solution qui s'offre à nous pour borner cette boucle est de sommer les

maximums d'itérations de la boucle (40 et 60) pour borner le nombre total d'itérations à 100. Ainsi, on indiquerait que dans l'ensemble du programme, la boucle n'itérera pas plus de cent fois.

```
void f(int k){
  for (int i=0; i<k; i++)
  factorielle(i);
}
int main(){
  f(40); // call 1
  f(60); // call 2
}
```

Code 3.1 – Deux appels à une fonction contenant une boucle.

En appliquant IPET, cette information globale laisse la liberté, lors de la résolution de l'ILP, de trouver la *pire* combinaison des deux boucles.

Dans notre programme, le fait de calculer la factorielle des entier de 1 à 100 sera bien plus coûteux que le cas réel qui calcule une première fois la factorielle des entier de 1 à 40 et une seconde fois de 1 à 60. Cet exemple illustre la nécessité dans certains cas d'avoir deux annotations locales plutôt qu'une seule globale.

Pour différencier les deux instances de la boucle `for`, la solution est de regarder le contexte d'appel (ou *call context*) qui aboutit à la boucle. Dans le code machine du programme principal `main`, les deux appels à la fonction `f` diffèrent par leur adresse physique. On peut donc envisager de fabriquer un automate qui se reposera sur ces appels pour fournir deux bornes de boucles contextuelles au lieu d'une borne globale.

3.5.2.2 Un premier automate avec contextes

Les automates à contraintes globales qui ont été présentés dans ce chapitre sont capables de supporter des annotations contextuelles relativement simples. La figure 3.10 montre un automate d'annotation supportant les informations de bornes de boucles du code 3.1 de manière contextuelle.

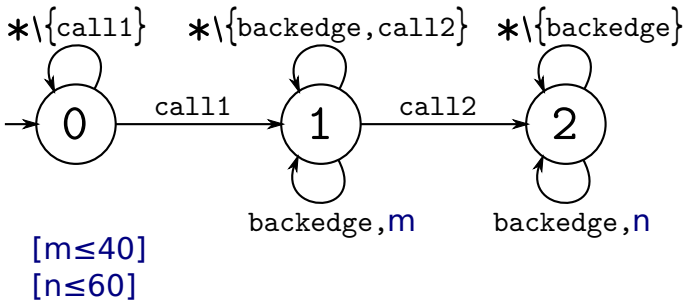


Figure 3.10 – FFA avec bornes de boucle contextuelles

Sur cette figure, on retrouve d'un côté deux contraintes `[m≤40]` et `[n≤60]` qui correspondent bien aux nombres maximums d'itérations de la boucle pour chaque appel

successif à `f` dans le programme principal. Ensuite, on trouve la variable `m` (resp.`n`) attachée à l'arc retour de la boucle (`backedge`) sur un état qui n'est atteignable que si on passe par la transition qui correspond au `call1` (resp. `call2`). De cette façon, si on fait un produit avec le CFA, on transférera les variables `m` et `n` en respectant les contextes d'appels `call1` et `call2`.

Limitations : Tout d'abord, pour fabriquer cet automate, il est nécessaire de connaître en plus des adresses de l'arc retour de la boucle, les adresses qui permettent d'identifier les différents appels de la fonction `f`. Ce problème n'est pas insurmontable et même relativement simple dans le cas qui nous intéresse, mais pour un programme plus complexe, l'automate d'annotation sera bien plus conséquent puisqu'il décrira l'intégralité des arbres d'appels, et qu'il faudra à ce moment là connaître toutes les adresses des différents appels de fonctions rencontrés.

D'autre part, la transition étiquetée `call2` joue deux rôles dans cet automate. Elle permet de définir d'une part l'entrée du contexte dans lequel on souhaite borner la boucle à soixante itérations, mais également la sortie du contexte précédent, dans lequel on souhaitait supporter la première borne locale (40). L'absence de transition de sortie pour le premier contexte est un problème en soi car, si dans le cas qui nous intéresse, l'entrée du second contexte va de pair avec la sortie du premier, ça n'est pas toujours le cas puisque les contextes peuvent également être imbriqués. L'entrée d'un second contexte ne pourra donc pas être utilisée systématiquement pour quitter un premier contexte. De plus, si on souhaite fabriquer séparément un automate pour chaque annotation contextuelle et qu'il n'existe pas de transition permettant de quitter l'état qui compte la borne de la première instance de la boucle ($\texttt{m} \leq 40$), on resterait alors indéfiniment dans cet état. La contrainte associée s'appliquerait alors sur le nombre total de `backedge` rencontrés depuis l'entrée dans ce contexte. Elle ferait ainsi office de contrainte globale et non plus locale.

3.5.2.3 Un second automate avec contextes

Cette seconde solution se base sur des informations de contexte plus fournies, à savoir non seulement les adresses des appels à la fonction `f`, mais aussi les arcs qui correspondent aux retours depuis cette fonction. Notons que la limitation concernant le trop grand nombre d'informations de contexte à connaître est encore plus présente ici puisqu'en plus des adresses des appels de fonctions, il faut connaître les adresses

correspondant aux retours des appels de fonctions [5].

La figure 3.11 montre deux FFAs séparés qui supportent chacun une annotation de borne de boucle contextualisée par un appel à la fonction `f`. Comme expliqué précédemment, sans l'ajout de la transition (0) ←return1— (1) au FFA (a), cette annotation s'appliquerait sur les deux instances de la boucle ce qui reviendrait à exprimer une borne globale de 40 itérations au total et qui serait faux.

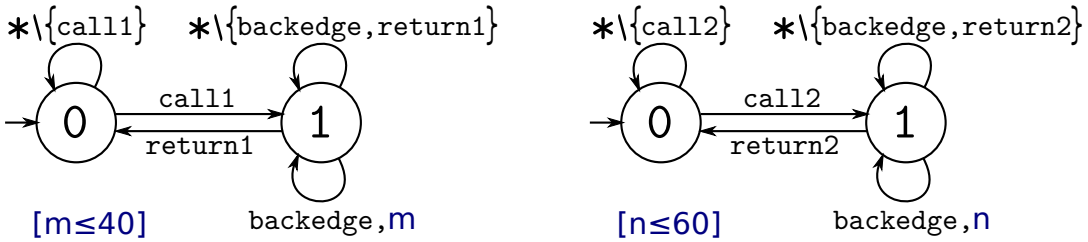


Figure 3.11 – FFAs supportant des annotations contextuelles de bornes de boucle.

Nous souhaitons à terme intégrer ces deux annotations dans le CFG. Pour cela, si l'on suit la démarche présentée dans ce chapitre, nous devons faire les produits des deux FFA (qu'on notera arbitrairement FFA_1 et FFA_2) et d'un CFA. L'opération de produit présentés dans la partie 3.4.5 est commutative, ce qui nous laisse la liberté de choisir l'ordre dans lequel nous ferons les produits. Il est donc possible d'envisager diverses solutions comme $(\text{CFA} \times \text{FFA}_1) \times \text{FFA}_2$, ou bien $(\text{CFA} \times \text{FFA}_1) \times (\text{CFA} \times \text{FFA}_2)$, ou encore $(\text{FFA}_1 \times \text{FFA}_2) \times \text{CFA}$. Cette dernière solution correspond bien à l'analyse de WCET, dans le sens où l'on intègre généralement depuis un fichier l'ensemble des annotations dans une analyse, et non pas une-à-une dans le CFG et le système ILP associé. Le résultat du produit des deux automates d'annotation de la figure 3.11 est présenté sur la figure 3.12.

On retrouve dans cette figure tous les comportements possibles combinés des deux automates de la figure 3.11. Les transitions (0) —call1→ (1) et (0) —call2→ (2) correspondent bien aux entrées dans la fonction `f` au travers des appels respectifs, et les transitions (0) ←return1— (1) et (0) ←return2— (2) capturent bien les sorties des deux contextes respectifs. De plus, dans l'état (1), on comptera bien le nombre de `backedge` dans le premier contexte, grâce à la contrainte [`m` $\leq$ `40`], tandis que l'état (2) permettra bien de borner la boucle dans le second contexte avec la contrainte [`n` $\leq$ `60`].

5. Nous verrons dans la partie 5.2 comment les informations de débogage peuvent être utilisées pour trouver quelles adresses physiques du programme correspondent aux différents éléments `call`, `return`, `loop`, etc.

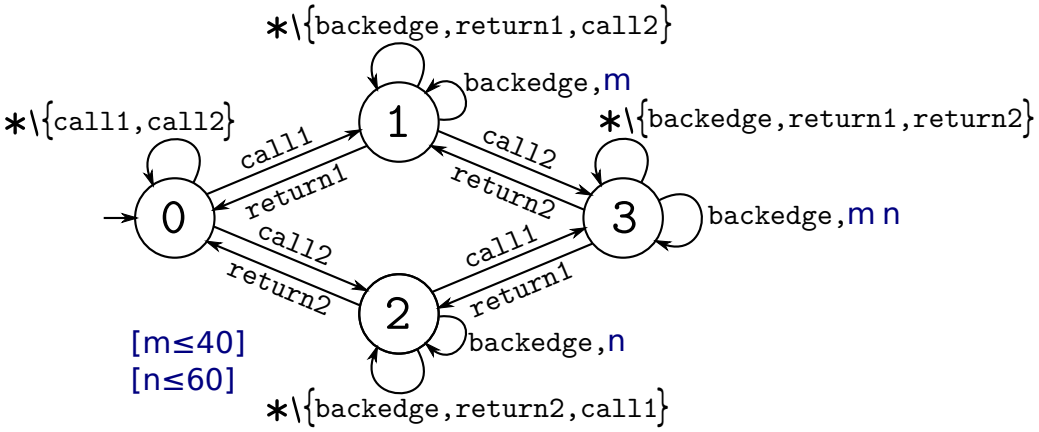


Figure 3.12 – FFA supportant des annotations contextuelles de bornes de boucle et résultant du produit des deux automates de la figure 3.11

Additionnellement, cet automate supporte l'imbrication des deux contextes d'appel à la fonction `f`, puisqu'il est issu d'un produit brut de deux automates, et que rien n'indique que cette imbrication de contextes n'a pas lieu dans le CFG réel. Il est ainsi possible d'entrer dans le second contexte à partir du premier et inversement, et d'arriver ainsi dans l'état (3). Dans cet état, le décompte de l'arc retour de la boucle doit impacter les deux contraintes à la fois, c'est pourquoi les deux variables `m` et `n` sont attachées à l'arc (3) `backedge` .

Une limitation fonctionnelle : la complexité

L'automate de la figure 3.12 supporte efficacement les deux annotations, que les contextes soient réellement imbriqués ou non. Cependant dans le cas illustré ici, lors du futur produit avec le CFA, l'état (3) ne sera jamais atteint puisqu'on quittera le premier contexte via (0) `return1` (1) avant d'entrer dans le second contexte. En d'autres termes, on peut considérer que la partie droite de l'automate (soit toutes les transitions liées à l'état (3)) est inutile. Inversement, si l'analyse portait sur un programme dans lequel le second contexte était bien imbriqué dans le premier, c'est la partie basse de l'automate (soit toutes les transitions liées à l'état (2)) qui aurait été inutile.

On en déduit que la méconnaissance des imbrications des contextes du programme cause un préjudice en terme de taille et de complexité d'automate. Dans le cas présenté ici, le nombre d'éléments superflus dans l'automate est relativement faible – un quart des états mais tout de même plus d'un tiers des transitions – mais pas négligeable

alors que le produit effectué implique seulement deux annotations et deux contextes. Si l'on considère le produit de 3 annotations, chacune dans un contexte, on obtient en résultat un graphe à 2^3 états, alors que le chemin réel dans le CFG n'impliquera tout au plus que 4 de ces états. Parallèlement, si on considère le produit de 2 annotations, chacune imbriquée dans un contexte et un sous-contexte (les automates pour chaque annotation seront donc composés de 3 états), on obtient un automate à 2^4 états soit 16 états, alors que seuls 5 d'entre eux participeront au produit avec le CFG.

La complexité de l'automate résultant de cette façon de gérer les contextes est clairement explosive, à la fois en nombre d'annotations, mais également avec le nombre de contextes et sous-contextes. Or, dans le domaine de l'analyse de WCET, y compris dans les programmes de référence testés habituellement (la suite de programmes de Mälardalen [23] par exemple), il n'est pas rare de rencontrer des programmes pour lesquels sont générés ou fournies des dizaines, voire des centaines d'annotations, impliquant une importante hiérarchie de contextes et de sous-contextes.

Une limitation d'expressivité : la localité des contraintes

Cette gestion des contextes souffre également d'un problème d'expressivité puisqu'elle ne supporte pas la notion de contexte local avec réentrance. Considérons le code 3.2 qui comporte une boucle dans une fonction appelée à divers points d'un programme et dont la borne locale à la fonction est 10. Cela signifie que cette borne locale restreint le nombre d'itérations de la boucle à 10 pour chaque entrée dans la fonction `foo`. On souhaite par ailleurs définir une seule annotation plutôt qu'utiliser une annotation par appel de la fonction (parce que leur nombre est trop important, ou parce que certains de ces appels apparaissent dans des boucles par exemple).

La première solution qui s'offre à nous pour exprimer cette contrainte locale consiste à construire un automate dans lequel le contexte correspondant à la fonction serait dupliqué autant de fois que la fonction serait appelée, comme illustré sur la figure 3.14. On pourrait alors définir un compteur local (`m`, `n`, ...) pour compter les arcs retours sur les états correspondants à chaque appel de la fonction ((1), (3), ...) en les associant chacun avec une contrainte différente (`[m≤10]`, `[n≤10]`, ...). Le problème avec cette approche est que l'on ne connaît pas ici le nombre de fois où la fonction est appelée, et que par conséquent, il faudrait construire un automate avec un nombre d'états et de contraintes potentiellement infini, ce qui n'est pas envisageable.

```
int foo(){
  for (int i=0; i<10; i++){
    ...
  }
}
int main(){
  ...
  foo();
  ...
  foo();
  ...
}
```

Code 3.2 – Boucle dans une fonction appelée de multiples fois.

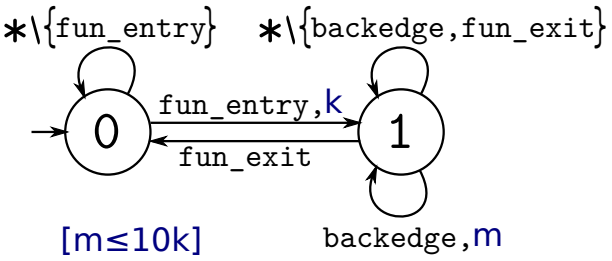


Figure 3.13 – FFA supportant une annotation de borne de boucle contextuelle grâce à un compteur supplémentaire.

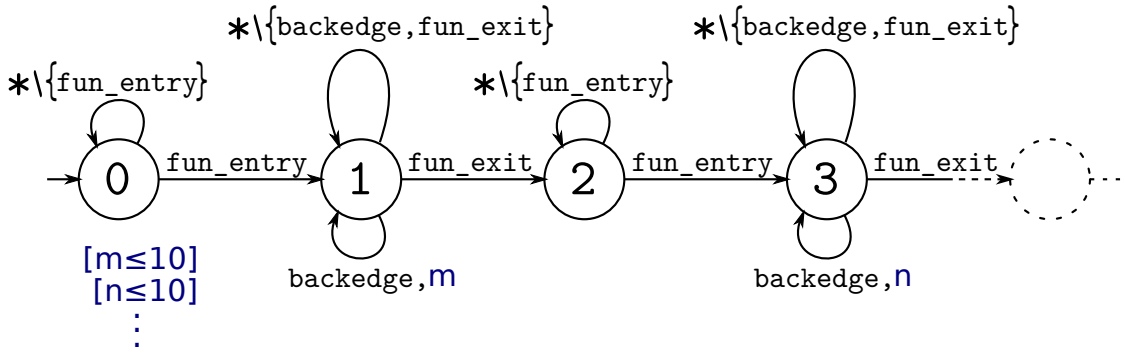


Figure 3.14 – FFA supportant une annotation de borne de boucle contextuelle par duplication des états.

Un seconde approche illustrée sur la figure 3.13 consiste à utiliser deux compteurs : l'un pour le décompte de l'arc retour de la boucle (`m`), et l'autre pour capturer le nombre d'entrées dans la fonction `k`. Ainsi, il est possible d'exprimer la contrainte `[m≤10k]` dérivée du fait que pour chaque entrée dans la fonction *foo*, l'arc retour peut être emprunté 10 fois. On a ainsi exprimé une contrainte globale à la place de la contrainte locale initiale mais on a perdu en précision par la même occasion. En effet, si la fonction est appelée deux fois dans le programme, la contrainte ainsi exprimée interdit simplement d'emprunter l'arc retour de la boucle plus de vingt fois au total. Ainsi, un chemin empruntant dix-neuf fois l'arc retour dans le premier appel, puis une seule fois dans le second est autorisée, alors qu'il ne l'était pas avec la borne locale, d'où la perte de précision.

3.6 Conclusion

Nous avons présenté dans ce chapitre un nouveau formalisme appelé automates à contraintes globales qui offre un support efficace à diverses annotations de flot. Nous avons montré comment il était possible d'intégrer les annotations portées par de tels automates dans un CFG et dans le système ILP qui lui est associé lors de l'estimation de WCET par la méthode IPET. La combinaison de plusieurs de ces automates a cependant fait apparaître les problèmes de complexité et d'expressivité de ce formalisme.

Dans certains langages d'annotation comme FFX, les informations de contextes seules représentent la structure du programme, dans laquelle les annotations de flot viennent se greffer. De cette façon, les contextes sont mutualisés et les diverses annotations peuvent être exprimées localement dans le contexte qui les englobe.

Afin de palier aux limitations mises en évidence en terme de complexité et d'expressivité, nous proposons de faire la même séparation dans nos automates et de construire une hiérarchie équivalente à, et représentative de l'imbrication des différents contextes du programme analysé. Ainsi les annotations seules pourront être définies localement à leur contexte, et la mise en commun de la hiérarchie de contextes réduira l'explosion du nombre d'états issue de la méconnaissance de la structure de ces contextes dans le programme réel.

"Don't you think if I were wrong, I would know it ?"

— Sheldon Cooper

4

Automates à contraintes hiérarchiques

Sommaire

4.1 Introduction

Nous avons montré dans la partie 3.4.1 qu'utiliser les adresses physiques des éléments du CFG était plus pertinent qu'utiliser la numérotation des blocs de base [1]. En contrepartie, il est possible de rencontrer un élément identifié par son adresse physique à

1. À cause de la non-unicité de la numérotation et l'impossibilité pour la numérotation de suivre les modifications du CFG.

différents endroits dans un même CFG. Chaque élément avec une même adresse physique est cependant différencié par son contexte d'appel, ce qui inclut les appels de fonctions et les itérations des boucles. On peut ainsi délimiter des sous-parties d'un programme par des contextes, dans lesquels il est possible d'exprimer des annotations de flot en isolation.

Il est possible de rendre les annotations de flot contextuelles de deux façons : soit en accompagnant chaque annotation d'informations sur son contexte, soit en décrivant l'arborescence des contextes du programme et en exprimant les annotations de flot à des endroits spécifiques de cette arborescence de façon à ce qu'elles soient appliquées localement. Dans le chapitre précédent, nous montré comment les automates à contraintes pouvaient supporter cette notion de contexte, mais nous avons également mis en évidence les problèmes de complexité et d'expressivité dont cette approche souffrait.

Nous proposons donc d'étendre le formalisme des FFAs présenté dans le chapitre 3 dans le but de supporter la notion de contexte séparément des annotations de flot en elles-mêmes. Pour cela nous ajoutons une structure hiérarchique aux automates, en autorisant les états à contenir eux-même des automates.

4.2 Idée générale

Dans l'article [46], nous présentons des automates hiérarchiques inspirés des *Statecharts* [26] adaptés au support d'annotation de flot contextuel. L'idée est d'exprimer les contextes d'un programme par des *super-états* capables de contenir d'autres automates. Arriver sur un super-état (via une transition entrante) permet ainsi d'entrer dans le contexte qu'il représente en empruntant la transition initiale du *sous-automate* qu'il contient. On sortira de ce contexte en suivant une transition sortante de ce super-état, tout en laissant le sous-automate dans un état *valide*, c'est-à-dire en vérifiant la validité de ses contraintes (voir partie 3.4.3).

L'utilisation d'une hiérarchie d'automates a pour but de suivre la structure des contextes d'appels des programmes, pour ainsi exprimer des annotations de flot locales à certains points précis des programmes. À terme, l'objectif reste d'utiliser ce support pour faciliter et améliorer l'intégration des annotations dans une analyse de WCET. Nous avons décrit dans le chapitre 3 un processus qui consistait à faire un produit entre CFA et FFA pour intégrer les annotations de flot dans la structure même du

CFG. Même si dans ce chapitre, les grandes étapes du processus resteront inchangées, l'utilisation d'automates hiérarchiques à la place des FFAs requiert néanmoins un certain nombre d'extensions pour parvenir à un résultat.

4.2.1 Des automates enrichis d'une hiérarchie

Dans un premier temps, nous aurons besoin de définir formellement une version enrichie des FFAs destinée à supporter les annotations contextuelles. Nous présenterons donc le formalisme des automates d'annotation de flot hiérarchiques ou *Hierarchical Flow fact Automata* (HFAs).

Les états des automates hiérarchiques pourront par définition contenir eux-même des automates. Cette structure impactera la fonction de transition des HFAs, puisqu'on évoluera à la fois entre les états, mais également en profondeur dans la hiérarchie. Dans la définition formelle de ces automates hiérarchiques, chaque état d'un HFA contiendra lui-même une liste de HFAs qui pourra être vide. Ainsi, les états qui contiendront une liste non-vide d'automates seront appelés des *super-états*, et les HFAs de cette liste seront appelés des *sous-automates*.

4.2.2 De nouvelles opérations

Dans le chapitre précédent, nous avons présenté une opération de produit entre deux automates à contraintes. Cette opération, bien qu'applicable uniquement sur des automates plats [2], convenait ainsi très bien au produit CFA $\times$ FFA. Cependant l'opération que nous cherchons désormais à effectuer doit pouvoir s'appliquer sur un automate plat (CFA) et un automate hiérarchique (HFA). Bien qu'il soit possible de simuler un FFA (et donc un CFA) avec un HFA [3], nous verrons que le produit de deux automates hiérarchiques pose des problèmes de cohérence en cas de chevauchement de contextes, c'est pourquoi nous ne présenterons pas le produit général entre deux HFAs.

À la place, nous exposerons un produit capable de faire l'*injection* d'un automate plat et d'un automate hiérarchique. Nous serons alors en mesure d'utiliser cette nouvelle opération pour injecter le CFA d'un programme dans un HFA contenant des annotations de flot contextuelles. Nous obtiendrons ainsi un nouvel HFA intégrant la

2. On parlera d'automate *plat* par opposition aux automates hiérarchiques de ce chapitre.
3. Les automates hiérarchiques que nous présentons étendent les FFA, donc il suffit d'utiliser un HFA qui n'aurait aucun super-état pour simuler un FFA.

structure du CFG qu'on nommera CFG hiérarchique. Néanmoins, les CFGs utilisés dans l'analyse de programmes sont des structures plates. C'est pourquoi nous proposons de définir une autre opération destinée à supprimer cette hiérarchie appelée *aplatissement*. Ainsi, il sera possible de reprendre la fin du processus présenté dans le chapitre 3 pour reconstruire le CFG et obtenir le WCET par la méthode IPET.

4.3 Définitions formelles

Cette partie décrit mathématiquement les automates à contraintes hiérarchiques, qu'on notera HFAs. Nous détaillerons également la fonction de transition étendue aux mots pour ce type d'automates, ainsi que le langage accepté par ces automates.

4.3.1 Automates hiérarchiques à contraintes

Ces automates étendent les FFAs présentés dans le chapitre 3 et diffèrent par l'ajout d'une fonction *sub* permettant de récupérer les sous-automates d'un super-état, et par l'adaptation de l'ensemble de contraintes Φ qui peut désormais porter sur des variables appartenant aux sous-automates.

Définition 9. *Un automate hiérarchique à contraintes est un tuple* $(N, i, \Sigma, sub, \mathcal{X}, \Phi, \delta)$ *tel que :*

- N *est un ensemble fini d'états,*
- $i \in N$ *est l'état initial de l'automate et* $sub(i) = \varepsilon$*,*
- Σ *est un alphabet fini,*
- $sub : N \to HFA^\star$ *est une fonction qui retourne une liste d'HFAs qui peut être vide,*
- $\mathcal{X}$ *est un ensemble de variables,*
- $\Phi : (\mathcal{X}_G(N, \mathcal{X}, sub) \to \mathbb{N}) \to \text{bool}$*, avec*

$$\begin{cases} \mathcal{X}_G\big(\mathcal{A}_i(N_i, i_i, \Sigma, sub_i, \mathcal{X}_i, \Phi_i, \delta_i)\big) = \mathcal{X}_G(N_i, \mathcal{X}_i, sub_i) \\ \mathcal{X}_G(N, \mathcal{X}, sub) = \mathcal{X} \uplus \big(\biguplus_{n \in N} \mathcal{X}_L(sub(n))\big) \\ \mathcal{X}_L(\mathcal{A}_1, \ldots, \mathcal{A}_k) = \biguplus_{i=1..k} \mathcal{X}_G(\mathcal{A}_i) \\ \mathcal{X}_L(\varepsilon) = \varnothing \end{cases}$$

est une conjonction de contraintes linéaires sur les variables de l'automate et des sous-automates récursivement.

- $\delta : N \times \Sigma \to N \cup \{\bot\} \times \mathcal{X}^\star$ *est une fonction de transition entre les états qui consomme une lettre de l'alphabet et renvoie l'état d'arrivée de la transition ou* $\bot$ *s'il n'en existe pas, ainsi qu'une liste de variables.*

4.3.2 Extension de la fonction de transition aux mots

Cette section présente l'extension aux mots de la fonction de transition δ définie sur les lettres pour les HFAs. La hiérarchie introduite dans ce chapitre rend cette transition appliquée aux mots plus complexe car elle joue un rôle plus important que dans le chapitre précédent. En effet, lorsqu'on souhaite consommer un mot avec un automate hiérarchique, certaines parties de ce mot risquent d'être consommées dans des contextes précis délimités par des super-états de l'automate courant. On parlera de *sous-chaîne* pour désigner les partie d'un mot qui seront consommées par des sous-automates plutôt que par l'automate courant.

Cette fonction de transition permet donc à la fois d'évoluer dans l'automate courant, mais également d'entrer dans des super-états, d'évoluer dans des sous-automates et de sortir des super-états en s'assurant que les sous-chaînes sont bien acceptées par le langage des sous-automates. De plus, les listes de variables rencontrées lors de la consommation des sous-chaînes sont également remontées lors des sorties de contextes. Enfin, cette fonction de transition permet la propagation d'erreurs depuis les sous-automates ou dans l'automate courant.

La complexité de cette fonction nous a poussé à séparer et détailler les différents cas pour faciliter la compréhension.

Définition 10. *Pour un HFA* $\mathcal{A}$*, la fonction de transition étendue aux mots s'applique sur un état, une liste de variables et un mot et renvoie un état et une liste de variables :*

$$\hat{\delta} : N \times \big(\mathcal{X}_G(\mathcal{A})\big)^\star \times \Sigma^\star \to N \times \big(\mathcal{X}_G(\mathcal{A})\big)^\star$$

On spécifie dans la suite l'ensemble des cas qu'il est possible de rencontrer lors de la consommation d'un mot.

Les différents cas présentés considèrent un automate hiérarchique à contraintes $\mathcal{A} = (N, i, \Sigma, \mathit{sub}, \mathcal{X}, \Phi, \delta)$, deux états $n, n' \in N$, une lettre $a \in \Sigma$, deux mots $\omega, \omega' \in \Sigma^\star$ et deux listes de variables $L, X \in \mathcal{X}^\star$. On détaillera le cas échéant la liste de sous-automates renvoyés par la fonction *sub*, et on écrira ε pour dénoter la liste vide.

On utilisera enfin la fonction *valid* ainsi que la définition du langage accepté par un HFA (partie 4.3.3 en page 57) pour vérifier l'acceptation des sous-chaînes par les sous-automates lors des sorties de contextes.

La transition a lieu uniquement dans l'automate courant : Ce cas correspond à celui des FFA présenté dans la partie 3.4.2 les états rencontrés ne contiennent pas de sous-automates. Ainsi, si la fonction de transition δ renvoie l'état n' en consommant la lettre a et que ni n ni n' ne sont des super-états, alors on se déplace simplement dans l'automate vers l'état n' en enrichissant la liste de variables L :

$$\hat{\delta}(n, L, a\omega) = \hat{\delta}(n', L \cdot X, \omega) \text{ si } \delta(n, a) = \langle n', X \rangle \wedge sub(n) = sub(n') = \varepsilon$$

Entrée dans un contexte : Une entrée de contexte correspond à l'arrivée sur un super-état par la consommation d'une lettre. Ainsi, si la fonction de transition δ renvoie le super-état n' en consommant la lettre a depuis l'état n, alors on se déplace dans l'automate en enrichissant L et en conservant la lettre a comme début de la sous-chaîne qui devra ensuite être consommée par le sous-automate :

$$\hat{\delta}(n, L, a\omega) = \hat{\delta}(n', L \cdot X, a\omega) \text{ si } \delta(n, a) = \langle n', X \rangle \wedge sub(n) = \varepsilon \wedge sub(n') \neq \varepsilon$$

Sortie d'un contexte : La sortie d'un contexte a lieu dès que l'on rencontre une lettre permettant de quitter le super-état. Pour cela, on découpe la chaîne en trois parties, en fonction de la première lettre rencontrée (notée ici a) permettant de sortir du super-état. La première partie correspond à la sous-chaîne qui devra être consommée par le sous-automate (ici ω, à laquelle il faut ajouter la dernière lettre a) tandis que la troisième partie (ici ω') correspond à la fin du mot qui devra être consommée ensuite.

La sortie d'un contexte est également régie par l'acceptation de la sous-chaîne dans les sous-automates. Cela implique de vérifier que la sous-chaîne à consommer (ωa) fait bien partie des langages acceptés par chaque sous-automate tout en remontant les variables rencontrées (dans x_t). La transition δ permet alors de changer d'état de l'automate courant :

$$\hat{\delta}(n, L, \omega a \omega') = \hat{\delta}(n', L \cdot x_t \cdot x, \omega') \text{ si } :$$

$$\begin{cases} \delta(n, a) = \langle n', x \rangle \wedge sub(n) \neq \varepsilon \wedge sub(n') = \varepsilon \\ \forall c \in \omega, \delta(n, c) = \langle \bot, \varepsilon \rangle \\ \forall\ \mathcal{A}_i = (N_i, i_i, \Sigma, sub_i, \mathcal{X}_i, \Phi_i, \delta_i) \in sub(n), \hat{\delta}(i_i, \varepsilon, \omega a) = \langle n_i, x_i \rangle \wedge valid(x_i, \Phi_i) \\ \text{ avec } n_i \in N_i \text{ et } x_i \in \left(\mathcal{X}_G(\mathcal{A}_i)\right)^\star \\ x_t = \underset{i}{concat}(x_i) \end{cases}$$

Sortie d'un contexte et entrée dans un contexte simultanées : Une même lettre peut permettre de sortir d'un contexte, d'emprunter une transition de l'automate courant puis d'entrer dans un nouveau contexte (ou le même). Il convient alors de combiner la règle de sortie de contexte avec la règle d'entrée de contexte. En premier lieu, on vérifie l'acceptation de la sous-chaîne ωa par les sous-automates de n en remontant les variables rencontrées, puis on évolue dans l'automate courant en conservant la lettre a comme début de la sous-chaîne qui devra être consommée dans les sous-automates de n' :

$$\hat{\delta}(n, L, \omega a \omega') = \hat{\delta}(n', L \cdot x_t \cdot x, a\omega') \text{ si } :$$

$$\begin{cases} \delta(n, a) = \langle n', x \rangle \wedge sub(n) \neq \varepsilon \wedge sub(n') \neq \varepsilon \\ \forall c \in \omega, \delta(n, c) = \langle \bot, \varepsilon \rangle \\ \forall\ \mathcal{A}_i = (N_i, i_i, \Sigma, sub_i, \mathcal{X}_i, \Phi_i, \delta_i) \in sub(n), \hat{\delta}(i_i, \varepsilon, \omega a) = \langle n_i, x_i \rangle \wedge valid(x_i, \Phi_i) \\ \text{ avec } n_i \in N_i \text{ et } x_i \in \left(\mathcal{X}_G(\mathcal{A}_i)\right)^\star \\ x_t = \underset{i}{concat}(x_i) \end{cases}$$

Notons que la seule différence avec le cas de sortie d'un contexte correspond à la présence du mot $a\omega'$ dans le résultat de la fonction $\hat{\delta}$.

Cas du mot vide : La consommation du mot vide correspond au cas d'arrêt. On renvoie dans ce cas l'état courant de l'automate et la liste de variables courante.

$$\hat{\delta}(n, L, \varepsilon) = \langle n, L \rangle$$

Propagation d'erreur par absence de transition : Si la fonction de transition δ ne permet pas de consommer la lettre a depuis l'état courant n, on propage alors une

erreur en renvoyant le couple $\langle \perp, \varepsilon \rangle$.

$$\hat{\delta}(n, L, a\omega) = \langle \perp, \varepsilon \rangle \text{ si } \delta(n, a) = \langle \perp, \varepsilon \rangle$$

Propagation d'erreur lors d'une sortie de contexte : Si l'état courant de l'automate est un super-état et qu'une erreur apparaît lors de la consommation de la sous-chaîne par un des sous-automate alors on propage cette erreur en renvoyant le couple $\langle \perp, \varepsilon \rangle$. Lors d'une sortie de contexte, une erreur peut apparaître dans un des sous-automate pour deux raisons : soit parce qu'une lettre de la sous-chaîne n'a pas pu être consommée et que la paire $\langle \perp, \varepsilon \rangle$ a été propagée, soit parce que la liste de variables x_i après consommation complète de la sous-chaîne ne valide pas les contraintes Φ_i de cet automate.

$$\hat{\delta}(n, L, \omega a \omega') = \langle \perp, \varepsilon \rangle \text{ si}$$

$$sub(n) \neq \varepsilon \wedge \exists\ \mathcal{A} \in sub(n) \text{ tel que } \omega a \notin \mathcal{L}(\mathcal{A})$$

Notons qu'on ne s'intéresse pas ici au résultat de la fonction de transition δ dans l'automate courant, puisqu'on propage l'erreur dans tous les cas.

Propagation d'erreur lorsque le mot n'est qu'une sous-chaîne : Si le mot à consommer ne permet pas de sortir du super-état courant. Cela signifie qu'on se trouve encore dans un contexte (et donc dans un sous-automate), duquel on ne pourra pas ressortir. Ce cas est considéré comme une erreur car l'ouverture d'un contexte va de pair avec sa fermeture, ce que ne permet pas le mot à consommer.

$$\hat{\delta}(n, L, \omega) = \langle \perp, \varepsilon \rangle \text{ si } :$$

$$sub(n) \neq \varepsilon \wedge \forall c \in \omega, \delta(n, c) = \langle \perp, \varepsilon \rangle \text{ avec } \omega \neq \varepsilon$$

La figure 4.1 illustre ce cas d'erreur avec un HFA qui accepte le mot *abcd* mais rejette le mot *abc*. En effet, on entre dans le super-état $\boxed{1}$ avec la lettre a et on consomme bien les lettres *abc* dans le sous-automate mais on ne sort pas de ce contexte.

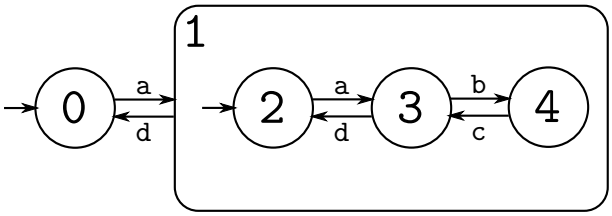


Figure 4.1 – Exemple d'automate hiérarchique à contrainte qui accepte le mot *abcd* mais rejette le mot *abc*.

4.3.3 Langage accepté par un automate hiérarchique

Le langage accepté par un automate correspond à l'ensemble des mots qu'il reconnaît. On définit le langage accepté de la même façon que pour les automates à contraintes non hiérarchiques, mais en se basant cette fois sur la fonction de transition étendue aux mots définie pour les automates hiérarchiques dans la partie 4.3.2. On utilisera par ailleurs la fonction *valid* qui vérifie qu'une liste de variables valide un jeu de contrainte présentée dans la partie 3.4.3.

Définition 11. *Soit $\mathcal{A}$ un automate hiérarchique à contraintes $(N, i, \Sigma, sub, \mathcal{X}, \Phi, \delta)$. Le langage accepté par cet automate noté $\mathcal{L}(\mathcal{A})$ est défini ainsi :*

$$\mathcal{L}(\mathcal{A}) = \left\{ \omega \in \Sigma^\star \mid \hat{\delta}(i, \varepsilon, \omega) = \langle n, L \rangle \wedge valid(L, \Phi) \text{ avec } n \in N \text{ et } L \in \left(\mathcal{X}_G(\mathcal{A})\right)^\star \right\}$$

En d'autres termes, un mot peut être rejeté pour deux raisons :

- si la fonction $\hat{\delta}$ a renvoyé le couple $\langle \bot, \varepsilon \rangle$ suivant un des cas d'erreur mentionnées dans la partie 4.3.2.
- si la liste de variables retournée ne satisfait pas le jeu de contraintes Φ, c'est-à-dire si *valid* a renvoyé *faux*.

4.4 Opérations

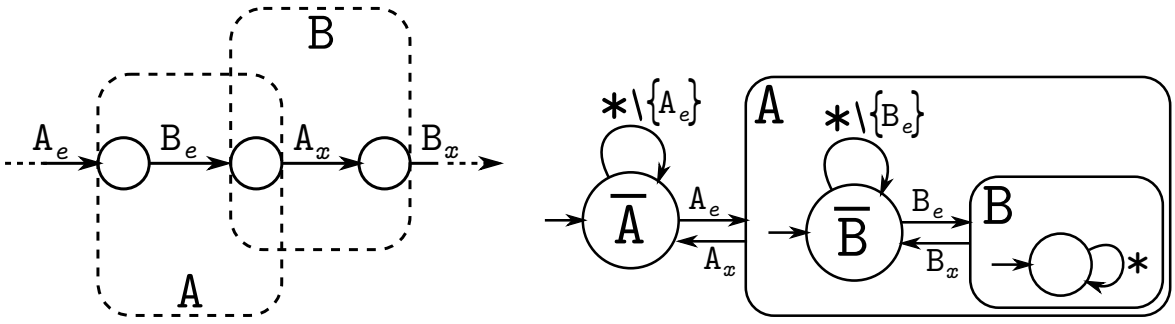
Nous présentons dans cette section les différentes opérations nécessaires pour intégrer des annotations portées par des HFAs dans le CFG d'un programme. Nous détaillerons en premier lieu les raisons qui nous empêchent de définir une opération de produit entre deux HFAs. Dans la suite de cette partie, une opération d'injection permettant de faire la fusion d'un FFA avec un HFA sera présentée, ainsi qu'une opération d'aplatissement destinée à transformer un HFA en un FFA. On présentera dans un premier

temps l'opération d'aplatissement car elle est plus simple appréhender et qu'elle permet d'introduire un certain nombre de fonctions et de notions qui seront également applicables à l'opération d'injection.

4.4.1 Un mot sur le produit de deux HFAs

L'idée de définir une opération de produit sur deux HFAs a été écartée à cause de la structure des automates hiérarchiques. En effet, par construction, le chevauchement de contextes n'est pas envisageable puisque la sortie d'un super-état *force* la sortie des contextes qu'il contient. Si l'on entre dans un contexte `A` puis un contexte `B`, il est impossible de sortir du contexte `A` et de rester dans le contexte `B` dans le même temps.

La figure 4.2a illustre ce problème. La transition d'entrée du contexte `B` par $\mathtt{B}_e$ a lieu dans le contexte `A` mais la transition de sortie du contexte `A` par $\mathtt{A}_x$ apparaît dans le contexte `B`. Intuitivement, on serait tenté de construire le HFA de la figure 4.2b, qui respecte bien les entrées et sorties de chaque contexte. Cependant, le mot $\mathtt{A}_e\mathtt{B}_e\mathtt{A}_x\mathtt{B}_x$ n'est pas accepté par cet automate : une fois entré dans les deux contextes imbriqués grâce aux lettres $\mathtt{A}_e\mathtt{B}_e$, la transition de sortie $\mathtt{A}_x$ qui devrait permettre de sortir du super-état `A` ne permettra pas de sortir du super-état `B` et une erreur sera alors propagée.



(a) Chevauchement de contextes. **(b)** HFA.

Figure 4.2 – Illustration du problème de chevauchement de contexte.

Théorème 1. *Le produit de deux HFAs ne peut pas être défini si certains de leurs contextes sont susceptibles de se chevaucher.*

Démonstration. Considérons deux contextes qui se chevauchent, c'est-à-dire que l'entrée du second a lieu dans le premier et que la sortie du premier a lieu dans le second. Il est possible de définir séparément un HFA pour chaque contexte, mais s'il était possible de définir leur produit qui résulterait en un troisième HFA équivalent, alors celui-ci

devrait nécessairement supporter le chevauchement de contexte, or la définition des HFAs ne le permet pas. □

Les problèmes rencontrés relatifs à ces chevauchement de contexte nous ont poussés à reprendre l'objectif initial de la définition d'un tel produit. Le but était la fusion d'un CFA représentant les informations de flot du programme et d'un HFA contenant des annotations supplémentaires. Or le CFA est nécessairement plat, donc au lieu de définir ce produit de HFAs, il suffit de définir un produit entre un FFA et un HFA qui ne souffrira pas des mêmes problèmes de chevauchement puisque les automates plats ne supportent pas de contextes. Après avoir présenté l'opération d'aplatissement et introduit un certain nombre de fonctions, on détaillera cette opération d'injection capable d'intégrer un automate plat dans un automate hiérarchique.

4.4.2 Opération d'aplatissement

Le produit d'injection entre un CFA et un HFA crée un nouvel automate hiérarchique. À terme cependant, ce résultat doit être utilisé pour reconstruire un nouveau CFG sur lequel doit ensuite être effectuée l'analyse de WCET. Il convient donc de définir une opération capable de transformer un HFA en un FFA équivalent et à partir duquel le nouveau CFG pourrait être reconstruit.

On appellera *aplatissement* l'opération permettant de construire un FFA équivalent à un HFA donné. Cette opération utilise le produit d'automates à contraintes présenté dans la partie 3.4.5 ainsi qu'une nouvelle opération appelée *unification*, destinée à intégrer dans un automate donné le sous-automate contenu dans un de ses super-états.

Les différents super-états d'un HFA peuvent contenir eux-mêmes des listes de HFA, et ainsi de suite de façon récursive. Cependant, nous travaillons sur des automates finis, donc en descendant suffisamment profondément dans la hiérarchie, nous trouverons toujours un HFA sans super-états, qu'on pourra donc simuler avec un FFA. Le principe de cette opération d'aplatissement est donc, pour chaque super-état rencontré, d'utiliser l'opération de produit d'automates à contraintes sur sa liste de sous-automates, en ayant pris soin d'aplatir les sous-automates qui le nécessitaient [4](par un appel récursif). Il suffira ensuite d'utiliser l'opération d'*unification* sur le super-état en question pour intégrer son unique sous-automate dans l'automate courant.

4. Le produit d'automates à contraintes est défini sur des FFA.

Fonction 1 : aplatissement

Données : HFA $\mathcal{A} = (N, i, \Sigma, sub, \mathcal{X}, \Phi, \delta)$
Résultat : FFA
pour chaque $n \in N$ tel que $sub(n) \neq \varepsilon$ **faire**
  FFA $\mathcal{F}^0 =$ `aplatissement`$(sub(n)[0])$
  pour $i =$ taille$(sub(n)) - 1$; $i > 0$; $i--$ **faire**
    FFA $\mathcal{F}^i =$ `aplatissement`$(sub(n)[i])$
    $\mathcal{F}^0 =$ `produit`$(\mathcal{F}^0, \mathcal{F}^i)$ *// produit d'automates à contraintes plats*
  $sub(n) = \{\text{HFA}(\mathcal{F}^0)\}$ *// $\mathcal{F}^0$ devient le seul sous-automate de n*
  `unification`$(\mathcal{A}, n)$
retourner FFA$(\mathcal{A})$ *// $\mathcal{A}$ n'est plus hiérarchique*

Cette opération d'aplatissement est présentée sous forme de fonction sur l'algorithme 1. On y détaille notamment l'utilisation du produit d'automates à contraintes et les appels récursifs à la fonction d'aplatissement sur la liste des sous-automates, pour chaque super-état rencontré.

Les différents appels récursifs et les boucles de cet algorithme permettent de travailler uniquement avec des automates plats au moment du produit d'automates, ainsi que lors de l'utilisation à la procédure d'*unification*. En effet, celle-ci ne peut être appelée que sur un super-état contenant un FFA comme unique sous-automate.

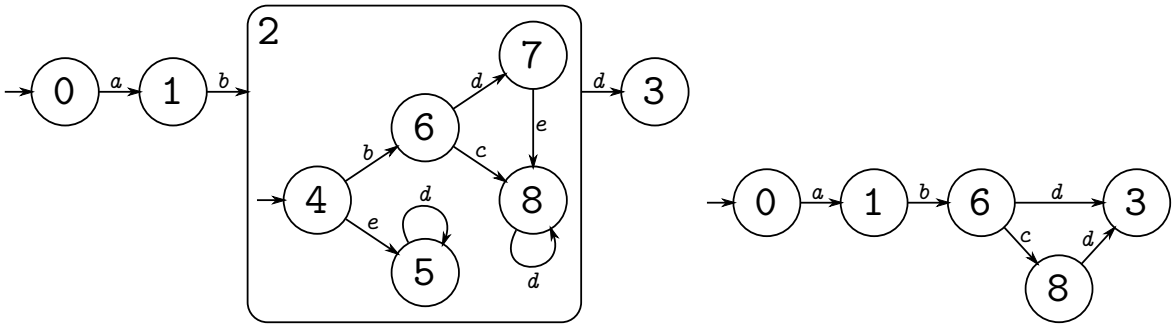
4.4.3 Unification : suppression des super-états

On appelle *unification* l'opération qui s'applique sur un super-état contenant un unique sous-automate non hiérarchique et qui consiste à supprimer ce super-état en intégrant le sous-automate à l'automate courant.

Les grandes étapes de l'opération d'*unification* appliquée au super-état n contenant un unique sous-automate plat sont les suivantes :

- Définir un automate $\mathcal{A}$ à partir des entrées et des sorties de n.
- Faire le produit de $\mathcal{A}$ avec le sous-automate de n.
- Intégrer le résultat du produit dans l'automate initial en accrochant ses transitions initiales aux transitions d'entrée dans n, et les transitions terminales aux transitions de sorties. On remplace ainsi le super-état n par une partie de son sous-automate.

La figure 4.4 illustre la suppression d'un super-état et l'intégration de son sous-automate à l'automate courant. La transition d'entrée du super-état $(1) \xrightarrow{b} \boxed{2}$ est mise



(a) HFA **(b)** HFA après *unification*

Figure 4.3 – Illustration de l'opération d'unification appliquée au super-état [2].

en correspondance avec la transition (4) $\xrightarrow{b}$ (6) du sous-automate puisque celle-ci a pour source l'état initial (4), tandis que la transition de sortie [2] $\xrightarrow{d}$ (3) est fusionnée avec les transitions (6) $\xrightarrow{d}$ (7) et (8) $\circlearrowleft_{d}$ du sous-automate, mais pas la transition (5) $\circlearrowleft_{d}$, car cette transition n'est pas atteignable en prenant comme première transition depuis l'état initial (4) celle qui respecte l'étiquette d'entrée `b`.

En effet, comme expliqué dans la partie 4.3.2 décrivant la fonction de transition étendue aux mots des automates hiérarchiques, les transitions d'entrée et de sortie d'un super-état ont une forme de priorité sur les transitions internes, et restreignent ainsi les mots acceptés par les sous-automates. Ainsi, pour qu'un mot soit accepté par un automate hiérarchique, il faut que les sous-automates des super-états rencontrés soient capables de consommer la *sous-chaîne* qui n'aura pas pu être consommée, préfixée de la lettre d'une transition d'entrée (ici `b`) et suffixée de la lettre d'une transition de sortie (ici `d`) de ces super-états.

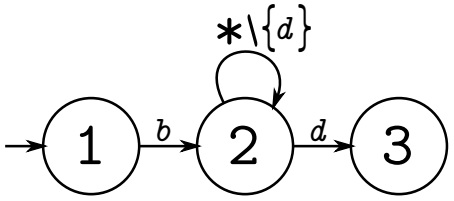
Cependant, il n'est pas impossible que le sous-automate seul accepte des mots ne respectant pas les étiquettes d'entrée et de sortie du contexte. Nous listons ici les différents cas et nous citons en exemple des mots acceptés par le sous-automate du super-état [2] de la figure 4.3a :

- Le mot ne commence pas par une lettre d'entrée du contexte. Ici, le sous-automate accepte le mot `ed` qui ne commence pas par `b`.
- Le mot ne termine pas par une lettre de sortie du contexte. Ici, le sous-automate accepte le mot `bc` qui ne termine pas par `d`.
- Le mot comporte une lettre de sortie ailleurs qu'en dernière position. Ici, le sous-automate accepte le mot `bded`, qui commence bien par `b` et termine par `d`, mais qui contient également un `d` en seconde position, ce qui autorise une sortie de contexte.

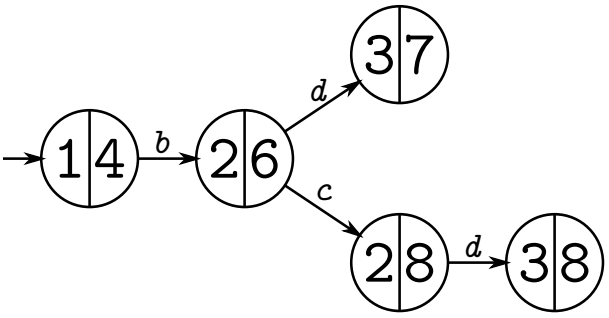
Il est donc nécessaire de réduire le sous-automate aux seuls états et transitions autorisant les mots voulus pour l'intégrer correctement dans l'automate englobant et que les langages acceptés par le HFA avant et après unification soient équivalents.

4.4.3.1 Réduction du sous-automate

Le moyen le plus simple de réduire ce sous-automate est de construire un automate forçant les transitions initiales et finales acceptées, et d'utiliser l'opération de produit. La figure 4.4a illustre un tel automate adapté de la figure 4.3a, qu'on appellera automate *réducteur*. On débutera la construction de cet automate par la création de son état initial à partir du prédécesseur du super-état. On ajoutera ensuite à l'automate un état pour chaque successeur du super-état, ainsi qu'un état pour le super-état en lui-même. On recopiera enfin à l'identique les transitions du HFA initial, et on ajoutera, sur l'état correspondant au super-état, une transition réflexive acceptant l'alphabet complet sauf les transitions sortantes. La figure 4.4b montre le résultat du produit entre le sous-automate du super-état $\boxed{2}$ et l'automate *réducteur* de la figure 4.4a.



(a) Automate *réducteur*.



(b) Résultat du produit entre l'automate *réducteur* et le sous-automate de $\boxed{2}$.

Figure 4.4 – Réduction du sous-automate du super-état $\boxed{2}$ par un produit forçant des transitions d'entrée et de sortie spécifiques.

4.4.3.2 Réinjection du résultat

Une fois le produit effectué, la dernière étape consiste à réintégrer son résultat à la place du super-état concerné par l'unification. Dans ce résultat, les états sont naturellement composés à partir des états du sous-automate et des états de l'automate *réducteur*. Il est ainsi possible de distinguer trois catégories d'états, en fonction des états de l'automate *réducteur* duquel ils proviennent :

1. Des états initiaux construits à partir des états initiaux des automates réducteurs. Ici l'état (1|4) provient du produit avec l'état (1).

2. Des états terminaux construits à partir des états terminaux des automates réducteurs. Ici les états (3|7) et (3|8) proviennent du produit avec l'état (3).

3. Des états issus spécifiquement du sous-automates, composés à partir des états autorisant toutes transitions dans les automates réducteurs. Ici les états (2|6) et (2|8) proviennent du produit avec l'état (2).

Le processus permettant de réintégrer le résultat dans l'automate initial consiste à remplacer d'un côté le super-état concerné par les états de la troisième catégorie, et inversement, de remplacer les états des deux premières catégories par les états desquels ils sont issus dans l'automate initial, en respectant les étiquettes correspondantes.

Dans notre exemple, les états (2|6) et (2|8) viennent remplacer le super-état [2], tandis que l'état (1|4) est remplacé par l'état (1), et que les états (3|7) et (3|8) sont remplacés par l'unique état (3). On retrouve ainsi l'automate de la figure 4.3b.

4.4.3.3 Adaptation des contraintes

Les HFAs définis dans la partie 4.3.1 supportent des jeux de contraintes associés à des variables attachées aux arcs. Les étapes de l'unification présentées dans la partie 4.4.3 montrent comment il est possible d'intégrer un sous-automate à la place du super-état qui le contenait, mais uniquement au niveau des états et des transitions. Nous avons repoussé à la présente section l'impact sur les contraintes de telles modifications d'un HFA, afin de simplifier les explications. Nous présentons donc dans cette partie les adaptations que nécessitent les contraintes lors de l'opération d'unification, et qui trouvent leur justification dans l'article [55]. Par ailleurs, on décrit ici un dépliage des contraintes pour chaque entrée dans un contexte, dans le seul but d'expliquer le fonctionnement général. Le dépliage est en réalité virtuel et ne nécessite pas d'être fait de façon explicite.

Les contraintes des HFAs sont vérifiées lorsqu'un mot ou une sous-chaîne doit être consommé en entier, par l'utilisation de la fonction *valid* (voir partie 3.4.3). En d'autres termes, à chaque fois qu'on sort d'un contexte, les occurrences des variables rencontrées doivent permettre de vérifier les contraintes associées.

Considérons un sous-automate supportant deux variables a et b et auquel la contrainte $a+2b \leq 8$ est associée. On peut assimiler les différentes entrées dans le contexte contenant ce sous-automate à différentes instances dans lesquelles les occurrences des variables doivent permettre de vérifier cette contrainte. On recopie ainsi virtuellement la contrainte autant de fois qu'on entre dans le contexte en question.

Pour chaque entrée dans le contexte, renommons les variables et leurs apparitions dans la contrainte associée avec un indice différent afin d'assurer leur indépendance. On utilisera des indices allant de 1 à n pour les n entrées dans le contexte, que l'on va de plus représenter explicitement au moyen d'une nouvelle variable n sur l'arc d'entrée de ce contexte [5]. Il est désormais possible de regrouper les différentes instances de cette même contrainte sous forme de système ainsi :

$$\begin{cases} a_1 + 2b_1 \leq 8 \\ a_2 + 2b_2 \leq 8 \\ \vdots \\ a_n + 2b_n \leq 8 \end{cases}$$

Lors de la phase de suppression d'un super-état, on remplace celui-ci par le sous-automate qu'il contient. Le nombre d'entrées dans le contexte correspond toujours à la variable n, ainsi qu'au nombre de fois où on a dupliqué la contrainte. Cependant, en remplaçant le super-état par son sous-automate, on ne sépare pas les différentes instances des variables supportées pas ce sous-automate, puisqu'on ne duplique pas n fois les arcs qui les supportent. On peut donc enrichir le système avec deux nouvelles équations pour exprimer la correspondance entre les différentes instances des variables dans les contraintes virtuellement dupliquées et l'unique instance des variables de l'automate réel :

$$\begin{cases} a = a_1 + a_2 + \cdots + a_n \\ b = b_1 + b_2 + \cdots + b_n \end{cases}$$

Afin d'exprimer la contrainte globalement et non plus localement au contexte dans lequel elle devait être vérifiée, il est maintenant nécessaire de sommer les différentes instances de la contrainte :

$$(a_1 + a_2 + \cdots + a_n) + 2(b_1 + b_2 + \cdots + b_n) \leq 8n$$

Pour finir on peut réécrire cette équation en fonction de a, b et n :

$$a + 2b \leq 8n$$

5. On utilise le nom n pour mettre en évidence le lien avec la borne de boucle en elle-même.

On a donc simplement multiplié les constantes par le nombre de fois où on est entré dans le contexte (la variable n), puisqu'on a sommé en réalité les n instances de la contrainte et opéré ensuite un renommage des variables.

Durant le processus de suppression de super-état, il a donc suffit de multiplier les constantes par le nombre n d'entrées dans le contexte. Pour cela, on placera systématiquement une variable sur l'arc d'entrée d'un super-état avant la phase d'unification, et on multipliera les constantes des contraintes par cette variable.

4.4.3.4 De l'indéterminisme pour les cas complexes

Il n'est pas impossible pour un contexte d'un HFA d'avoir de multiples transitions entrantes, ce qui pose un problème durant l'opération d'unification. En effet, chaque état dont une transition permet d'entrer dans le super-état auquel on s'intéresse doit devenir un état initial de l'automate *réducteur*, ce qui le rend alors indéterministe par construction. On ajoutera alors une ε-transition permettant d'accéder à chaque état initial avant de faire le produit avec le sous-automate. Le résultat comportera donc également cette ε-transition faisant l'union de tous ses états initiaux. Néanmoins, l'étape finale qui consiste à réintégrer ce résultat à la place du super-état concerné permettra de fusionner les différents états initiaux avec les états d'entrée de l'automate initial, ce qui éliminera l'indéterminisme.

L'adaptation des contraintes (partie 4.4.3.3) ne diffère que par le nombre d'entrées dans le contexte, qui dépend maintenant de la somme des entrées sur chaque arc. Il suffit cependant d'ajouter une seule et même variable sur tous les arcs d'entrée, qui viendra ensuite multiplier les constantes des contraintes, de la même façon que lorsque l'entrée est unique.

4.4.3.5 Algorithme de l'unification

La procédure 2 présente cette procédure d'unification. Dans cet algorithme, on utilisera notamment les fonctions suivantes :

- *reductAuto* (HFA,état) permet de fabriquer l'automate réducteur associé au super-état passé en paramètre, comme expliqué dans la partie 4.4.3.1.
- *associerDepuis* ([transition],[transition]) permet de faire la fusion de deux listes de transitions en remplaçant, pour toutes les paires de transitions portant les mêmes

étiquettes, l'état source de la seconde par l'état source de la première. Inversement, *associerVers* permet de faire cette fusion en remplaçant l'état destination de la seconde transition par l'état destination de la première.

- *adapterContraintes*(FFA,variable) multiplie les constantes des contraintes par une variable passée en paramètre, comme expliqué dans la partie 4.4.3.3.
- *pred*(état) permet de récupérer l'ensemble des transitions depuis les prédécesseurs d'un état passé en paramètre. Inversement, *succ* correspond aux transitions vers les successeurs de cet état.
- *produit*(FFA,FFA) correspond à l'opération de produit de deux automates à contraintes présentée dans la partie 3.4.5.

Procédure 2 : unification

Données : HFA $\mathcal{A} = (N, i, \Sigma, sub, \mathcal{X}, \Phi, \delta)$, $n \in N$
1 FFA $\mathcal{F}$ = *produit*(sub(n),*reductAuto*(n))
2 *associerDepuis*(*pred(n)*,première transition de $\mathcal{F}$) // *variable n_e incluse*
3 *associerVers*(*succ(n)*,transitions($\mathcal{F}$)) // *associe toutes les transitions de $\mathcal{F}$*
4 copier les états de $\mathcal{F}$ dans $\mathcal{A}$
5 copier les transitions de $\mathcal{F}$ dans $\mathcal{A}$
6 *adapterContraintes*($\mathcal{F}$, n_e)
7 copier les contraintes de $\mathcal{F}$ dans $\mathcal{A}$
8 supprimer n

L'algorithme présenté ici ne supporte pas l'indéterminisme mentionné dans la partie 4.4.3.4, mais pour l'adapter il suffirait de changer à la ligne 1 le type de $\mathcal{F}$ en un équivalent non déterministe, et de mettre "première transition" au pluriel à la ligne 2.

4.4.4 L'opération d'injection

Le langage $\mathcal{L}_1$ accepté par le CFA d'un programme représente l'ensemble des traces qui respectent la structure de ce programme, tandis que le langage $\mathcal{L}_2$ accepté par un HFA décrit l'ensemble des traces qui respectent l'annotation (ou les annotations) que cet automate hiérarchique supporte. Le résultat de l'opération d'injection qu'on souhaite définir doit être un automate dont le langage accepté soit $\mathcal{L}_1 \cap \mathcal{L}_2$.

Intégrer un FFA dans un HFA consiste à faire correspondre des parties de l'automate plat avec les contextes définis dans l'automate hiérarchique. Pour cela, on utilisera un certain nombre de fonction triviales de manipulation des automates, combinées dans un algorithme récursif qui renverra un HFA en résultat. Le fonctionnement de l'algorithme

est détaillé ci-après, suivi des fonctions utilisées.

Fonctionnement général : Lorsque les états concernés par le produit ne sont pas des super-état, le fonctionnement est le même que pour pour le produit d'automates à contraintes, c'est à dire qu'on évolue dans les deux automates en parallèle en construisant les états et les arcs en commun. Les différences apparaissent lorsque l'état de destination de la transition explorée est un super-état. Dans ce cas, l'algorithme *découpe* la partie du FFA qui sera précisément injectée dans ce super-état, et continue ensuite la propagation du produit à partir des états de sortie de ce super-état et des états qui suivent la découpe dans le FFA.

Le processus de découpe est le même que celui utilisé dans l'opération d'unification puisqu'il repose sur la fabrication de l'automate *réducteur*(voir partie 4.4.3.1) du super-état auquel on s'intéresse, puis sur l'utilisation du produit d'automates entre l'automate réducteur et le FFA. On prendra soin cependant de redéfinir l'état initial du FFA comme l'état courant afin d'obtenir précisément la portion du FFA qui devra être injectée récursivement dans les différents sous-automates du super-état.

La fonction 3 présente l'opération d'injection. Dans cet algorithme, les fonctions suivante seront utilisées :

- *produitPartiel*(FFA, état, FFA) permet de faire un produit entre deux FFAs, en considérant que l'état initial du premier FFA est l'état passé en paramètre.
- *last*(FFA) permet de rechercher, dans un automate, les états qui ne sont la source d'aucune transition, c'est-à-dire qu'il ne présentent aucun arc de sortie.
- *builtFrom*(FFA, état) renvoie l'état du FFA correspondant à l'état passé en paramètre. Il est nécessaire cependant que l'état passé en paramètre soit le résultat d'un produit d'automates impliquant le FFA en question. Cette fonction permet de remonter à l'état originel à partir duquel l'état passé en paramètre a été construit.
- *builtIn*(FFA, état) renvoie la liste des états du FFA qui ont été construits à partir de l'état passé en paramètre.

On utilisera également la fonction *reductAuto*(HFA,état) déjà définie dans la partie 4.4.3.5.

Fonction 3 : injection

Données : FFA $\mathcal{F}$, HFA $\mathcal{H}$
Résultat : HFA $\mathcal{R}$

1 *//On considère une liste de travail WL contenant trois états*
2 Initialiser le nouvel HFA $\mathcal{R}$
3 Créer l'état initial $\mathcal{R}_0 = (\mathcal{F}_0, \mathcal{H}_0)$ et l'ajouter à $\mathcal{R}$
4 Initialiser *WL* avec les trois états initiaux $\langle \mathcal{F}_0, \mathcal{H}_0, \mathcal{R}_0 \rangle$
5 **pour chaque** couple $\langle \mathcal{F}_i, \mathcal{H}_i, \mathcal{R}_i \rangle$ de *WL* **faire**
6 **pour chaque** α tel que $\delta_{\mathcal{F}}(\mathcal{F}_i, \alpha) = \langle \mathcal{F}'_i, x_{\mathcal{F}} \rangle$ **faire**
7 **si** $\delta_{\mathcal{H}}(\mathcal{H}_i, \alpha) = \langle \mathcal{H}'_i, x_{\mathcal{H}} \rangle$ **alors**
8 **si** $\mathcal{H}'_i$ est un super-état **alors**
9 Créer $\mathcal{R}'_i = (\mathcal{F}_i, \mathcal{H}'_i)$ et l'ajouter à $\mathcal{R}$
10 Construire FFA $C_{(\mathcal{H}'_i)} =$ `reductAuto`$(\mathcal{H}, \mathcal{H}'_i)$
11 Construire FFA $P_{(\mathcal{F}_i)} =$ `produitPartiel`$(\mathcal{F}, \mathcal{F}_i, C_{(\mathcal{H}'_i)})$
12 **pour chaque** sous-automate $sub^{\mathcal{H}}$ de $\mathcal{H}'_i$ **faire**
13 Créer HFA $sub^{\mathcal{R}} =$ `injection`$(P_{(\mathcal{F}_i)}, sub^{\mathcal{H}})$
14 Ajouter $sub^{\mathcal{R}}$ à la liste des sous-automates de $\mathcal{R}'_i$
15 **pour chaque** état C_j de la liste `last`$(C_{(\mathcal{H}'_i)})$ **faire**
16 Retrouver $\mathcal{H}_j =$ `builtFrom`$(\mathcal{H}, C_j)$
17 Trouver la lettre β telle que $\delta_{\mathcal{H}}(\mathcal{H}'_i, \beta) = \langle \mathcal{H}_j, x'_{\mathcal{H}} \rangle$
18 **pour chaque** état P_j de la liste `builtIn`$(P_{(\mathcal{F}_i)}, C_j)$ **faire**
19 Retrouver $\mathcal{F}_j =$ `builtFrom`$(\mathcal{F}, P_j)$
20 Créer $\mathcal{R}_j = (\mathcal{F}_j, \mathcal{H}_j)$ et l'ajouter à $\mathcal{R}$
21 Ajouter $(\mathcal{F}_j, \mathcal{H}_j, \mathcal{R}_j)$ à *WL*
22 **pour chaque** état $\mathcal{F}_k$ tel que $\delta_{\mathcal{F}}(\mathcal{F}_k, \beta) = \langle \mathcal{F}_j, x'_{\mathcal{F}} \rangle$ **faire**
23 Ajouter un arc $(\mathcal{R}'_i) \xrightarrow{\beta,\, x'_{\mathcal{F}} \uplus x'_{\mathcal{H}}} (\mathcal{R}_j)$
24 **sinon**
25 Créer $\mathcal{R}'_i = (\mathcal{F}'_i, \mathcal{H}'_i)$ et l'ajouter à $\mathcal{R}$
26 Ajouter $(\mathcal{F}'_i, \mathcal{H}'_i, \mathcal{R}'_i)$ à *WL*
27 Ajouter un arc $(\mathcal{R}_i) \xrightarrow{\alpha,\, x_{\mathcal{F}} \uplus x_{\mathcal{H}}} (\mathcal{R}'_i)$
28 **retourner** $\mathcal{R}$

4.5 Conclusion

Dans ce chapitre, nous avons étendu le formalisme des FFAs présenté dans le chapitre précédent, en introduisant la notion de hiérarchie d'automates et en définissant le formalisme des HFAs. Les états d'un tel automate peuvent eux même contenir une liste d'automates hiérarchiques, et seront utilisés pour représenter les contextes de validité

des annotations de flots portées habituellement par des langages d'annotation.

Les opérations nécessaires à l'intégration des ces automates hiérarchiques dans une analyse de WCET ont également été définies. Dans le chapitre précédent, l'approche permettant d'intégrer les annotations portées par les FFAs dans le CFG reposait sur l'utilisation du produit d'automates entre un CFA et un FFA porteur d'annotations. L'utilisation de HFAs à la place des FFAs rendait impossible l'utilisation du produit d'automates classique. Nous avons donc défini les diverses opérations nécessaires pour obtenir une fusion efficace d'un CFA et d'un HFA, et pour retrouver le résultat de cette fusion sous forme non hiérarchique, afin de pouvoir reconstruire le CFG correspondant et être en mesure de l'analyser.

Le chapitre 5 illustrera l'utilisation de ces HFAs, puisque l'on en construira pour les différents contextes des annotations liées à l'analyse de WCET. Les opérations associées au processus d'intégration seront également mentionnés, et leur utilisation sera détaillée dans le chapitre 6.

"Just remember every time you look up at the moon, I too will be looking at a moon. Not the same moon, obviously, that's impossible."

— Andy Dwyer

5

Application au langage d'annotation FFX

Sommaire

5.1 Introduction

Nous avons présenté dans les chapitres précédents des structures basées sur les automates capables de supporter des contraintes et des contextes, et des opérations pour fusionner automates d'annotation et graphes de flot de contrôle offrant ainsi une alternative aux langages d'annotation pour intégrer des propriétés sémantiques dans

un processus d'analyse de WCET. Dans ce chapitre, nous montrerons comment il est possible de transformer les différents éléments du langage d'annotation FFX en automates d'annotations. Une des difficultés rencontrées est l'obtention, grâce aux informations de débogage, des adresses physiques qui correspondent aux différents éléments qui peuvent composer un programme. Nous verrons ainsi comment construire des automates hiérarchiques compacts correspondant aux différents contextes d'un programme, pour pouvoir ensuite intégrer des annotations contextuelles dans une analyse de WCET. Nous appliquerons enfin ces diverses constructions lors d'une traduction depuis le langage d'annotation FFX.

Un premier exemple de fichier FFX

Nous présentons dans cette partie un exemple simple de fichier FFX généré à partir du code source d'un programme écrit en langage C et l'interprétation qui peut en être faite.

```
int foo(int n){
  int j=0,k;
  for(k=0; k<10; k++){
    j+=n-k;
  }
  return j;
}
int main(){
  foo(5);
}
```

Code 5.1 – Programme en langage C.

```
<flowfacts>
 <function name="main">
  <call name="C1" source="test.c" line="9">
   <function name="foo">
    <loop source="test.c" line="3" maxcount="10" >
    </loop>
   </function>
  </call>
 </function
</flowfacts>
```

Code 5.2 – Fichier FFX généré à partir du code 5.1.

Le langage d'annotation FFX décrit dans la partie 2.4.1 permet d'exprimer des contraintes de flot sur un programme donné. Le code 5.2, par exemple, exprime la borne de la boucle qu'on trouve dans la fonction `foo` du programme 5.1. Pour cela, la borne de la boucle est définie comme un attribut (`maxcount="10"`) de la balise de boucle (`<loop>`). Les autres attributs de cette balise permettent de localiser la boucle dans le programme, au moyen du nom du fichier source et de la ligne à laquelle elle se trouve. On peut également noter la hiérarchie des balises qui traduit la structure du programme en elle-même : la balise de boucle est imbriquée dans une balise de fonction, elle-même contenue dans une balise correspondant à son appel que l'on peut trouver à la ligne 9 du fichier source, et puisque cet appel de fonction se situe dans le

programme principal, cette balise <`call`> est contenue dans la balise correspondant à la fonction `main`. Pour finir, on retrouve la balise <`flowfacts`> à la racine du fichier FFX.

5.2 Du code source vers le fichier binaire

Certains langages d'annotations (FFX notamment) supportent l'expression de propriétés à la fois sur le code source des programmes, mais également directement sur leur binaire. Lorsqu'un expert fournit des informations supplémentaires sur un programme, il utilisera naturellement des annotations rapportées à des lignes précises dans le code source où les structures sont bien identifiées, tandis que les annotations générées automatiquement par un outil pourront aussi bien reposer sur les adresses physiques que sur les lignes des fichiers source. Ainsi, on pourra par exemple trouver une borne de boucle avec comme adressage la ligne précise du fichier où trouver les mots-clés *for* ou *while*, ou à l'inverse, directement l'adresse physique du bloc correspondant à la tête de cette boucle.

À terme, dans le CFG, on ne trouvera que des adresses physiques, si bien que lorsque les annotations se rapportent au code source et sont donc identifiées par des lignes des fichiers sources, il est nécessaire de traduire ces annotations en adresses physiques pour faire le rapprochement avec le CFG.

On utilise pour cela les informations de *débogage* d'un programme qui permettent de faire le lien entre les lignes du fichier source et les adresses physiques qu'on trouve dans le fichier binaire issu de la compilation. On obtient ainsi une forme de traçabilité des informations depuis le fichier source jusqu'au fichier binaire, ce qui n'est pas un problème trivial et a fait l'objet de multiples recherches [37, 34, 30].

Par ailleurs, la mise en correspondance des numéros de lignes d'un fichier source avec les adresses physiques du binaire impose de n'avoir qu'une instruction par ligne dans le fichier source, et ce afin de s'assurer que deux informations de débogages relatives à deux instructions différentes ne renvoient pas vers un seul et même numéro de ligne. En effet, si sur une même ligne par exemple, on venait à trouver deux appels à deux fonctions différentes, les informations de *débogage* ne suffiraient pas à différencier les appels et d'autres données comme le nom de la fonction appelée devraient être utilisées pour faire cette distinction.

La méthode que nous présentons dans cette thèse consiste à transformer les annotations en automates afin de les intégrer dans le CFG. L'alphabet utilisé pour ces automates est donc basé sur des informations issues du CFG, en l'occurrence les adresses physiques des blocs de base. Ainsi, la correspondance entre les informations sur le code source et les adresses physiques du binaire nous est nécessaire dès la fabrication de ces automates d'annotation.

5.2.1 L'alphabet des automates basé sur le CFG

Dans la partie 3.2.1 nous avons montré comment il était possible de transformer un CFG en un automate qu'on a nommé CFA, en utilisant comme alphabet les couples d'adresses ⟨source, destination⟩ correspondant aux adresses des blocs de bases auxquels sont rattachées les arcs.

Nos automates sont destinés à autoriser des ensembles spécifiques d'arêtes d'un CFG, et par complément à en interdire d'autres. Il est donc indispensable d'utiliser le même alphabet que celui du CFA qu'on dérivera de ce CFG. De cette façon on pourra utiliser les opérations d'injection et de produit sur un alphabet commun. Or dans certains cas, on ne s'intéressera qu'au bloc source ou au bloc destination d'un arc, sans se soucier des autres blocs. Afin de conserver une description d'automates la plus compacte possible, nous avons donc décidé de définir une notation pour représenter un ensemble de transitions dont l'une ou l'autre des adresses du couple est quelconque.

5.2.2 L'utilisation de l'étoile

Nous utiliserons l'étoile ($*$) pour représenter la notion d'adresse quelconque, par le remplacement de l'une ou l'autre des adresses du couple ⟨source, destination⟩. Une transition étiquetée avec le couple $\langle \texttt{adr}_x, * \rangle$ représentera ainsi l'ensemble des transitions qui ont pour bloc source le bloc d'adresse $\texttt{adr}_x$, quelle que soit l'adresse du bloc destination.

On pourra ainsi trouver des transitions du type $\texttt{adr}_x \to *$, ou à l'inverse $* \to \texttt{adr}_x$. Notons que la transition étiquetée $* \to *$ est équivalente à la transition $*$ et accepte l'ensemble de l'alphabet Σ.

De plus, cette notation est compatible avec celle introduite dans la partie 3.4.1 qui autorise la syntaxe $*\backslash\{\langle \texttt{adr}_x, \texttt{adr}_y \rangle\}$ pour exprimer l'ensemble des transitions existantes

dans le CFG sauf les transitions $\texttt{adr}_x \to \texttt{adr}_y$. Il est donc possible de trouver des transitions comme $\langle \texttt{adr}_x, * \rangle \backslash \{\langle \texttt{adr}_x, \texttt{adr}_y \rangle\}$ qui représente donc l'ensemble des transitions qui partent du bloc d'adresse $\texttt{adr}_x$ sauf celle qui a pour destination le bloc d'adresse $\texttt{adr}_y$.

Si cette notation permet d'éviter les ambiguïtés elle a le désavantage d'être particulièrement verbeuse, notamment lors de la représentation des automates sous forme graphique. Ainsi, lorsque ça sera possible, on représentera les transitions en considérant une priorité entre les transitions, qui correspondra à la *précision* de la transition, c'est-à-dire à la taille de l'ensemble représenté :

- Priorité 1 : $\texttt{adr} \to \texttt{adr}$.
- Priorité 2 : $\texttt{adr} \to *$ ou $* \to \texttt{adr}$.
- Priorité 3 : $* \to *$ noté aussi $*$ ou encore Σ.

La figure 5.1a montre un automate avec trois transitions ainsi que les transitions exclues, qui ont été omises sur la figure 5.1b. Il convient donc d'interpréter ce second automate avec la notion de priorité. En l'occurrence la transition $\texttt{adr}_x \to \texttt{adr}_y$ est la plus précise, ce qui implique qu'elle est exclue de la transition $\texttt{adr}_x \to *$, qui est elle-même exclue de la transition $*$. En considérant cette interprétation, les deux automates sont équivalents.

D'autre part, le formalisme des automates présenté dans le chapitre 4 et utilisé ici est déterministe. Il est donc nécessaire de définir clairement cette règle de priorité pour que l'automate de la figure 5.1b ne puisse être interprété que de façon déterministe.

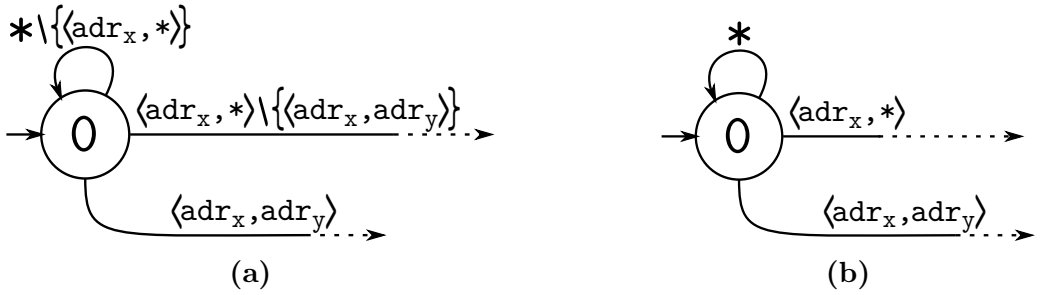


Figure 5.1 – Deux automates équivalents, l'un avec les transitions exclues explicites (a) et l'autre avec la notion de priorité (b).

Notons que deux transitions de type $\texttt{adr}_x \to *$ et $* \to \texttt{adr}_y$ ont la même priorité et nécessitent l'ajout explicite des transitions exclues pour éviter les ambiguïtés et conserver le déterminisme des automates.

5.3 Différents contextes

Dans cette partie, nous passons en revue les différents types d'informations issues du code source que nous souhaitons pouvoir supporter, et comment il est possible d'interpréter les informations de débogage qui leur sont associées pour construire des automates hiérarchiques aussi compacts que possible. Les contextes qui seront supportés correspondent aux constructions que l'on trouve dans les fichiers FFX et qui représentent les éléments structurels usuels de la programmation impérative.

5.3.1 Contexte d'appel de fonction

La modularité des programmes provient du découpage en fonctions, ce qui permet d'appeler une même fonction plusieurs fois depuis des points différents du programme. Si dans le corps d'une fonction, une propriété sémantique dépend des paramètres de cette fonction, alors il sera pertinent de fournir des annotations différentes en fonction du contexte d'appel, et non pas pour tous les appels à cette fonction. Dans la partie 3.5.2 par exemple, nous présentons un programme avec deux appels à la même fonction et une borne de boucle différente pour chaque appel.

Un appel de fonction peut être décomposé en deux parties : d'un côté l'appel en lui même qui correspondra au bloc de base depuis lequel le branchement d'appel est effectué, et de l'autre le corps de la fonction soit tous les blocs de base qui la composent.

L'appel : Un appel de fonction correspond à une instruction de branchement située à la fin d'un bloc de base. Ainsi, un appel de fonction depuis le programme principal correspondra à un seul et unique bloc de base dans un CFG. On travaille cependant sur le CFG *inliné* des programmes, ce qui implique que, comme illustré dans le code 5.3, si une fonction (ici `bar`) est appelée dans une fonction (ici `foo`) qui est elle-même appelée deux fois depuis le programme principal (`C1` et `C2`), alors l'appel auquel on s'intéresse (`C3`) correspondra à deux blocs de bases à deux endroits différents dans le CFG mais avec la même adresse physique. De plus, si on regarde uniquement le CFG de la fonction `foo` plutôt que le CFG du programme complet, on ne trouvera bien qu'un seul bloc correspondant à cet appel `C3`. Ainsi on a bien une correspondance directe entre

```
void foo(){
  bar(); // C3
}
int main(){
  foo(); // C1
  foo(); // C2
}
```

Code 5.3 – Appels de fonctions imbriqués

l'appel de fonction et les adresses physiques des blocs dans le contexte spécifique de l'appel, puisque de leur côté, les blocs de base des deux appels `C1` et `C2` à cette même fonction `foo` auront bien chacun une adresse physique différente.

D'autre part, dans le binaire d'un programme, le point d'appel d'une fonction correspondra à une instruction de saut (*branch*), tandis que l'instruction en séquence sera habituellement sauvegardée dans un registre spécifique (par exemple le registre *r14*, appelé *Link Register* en ARM) ou sur la pile pour pouvoir y revenir par la suite, puisqu'elle correspond justement à l'adresse de retour de cette fonction.

Ces deux adresses sont précisément celles nécessaires pour délimiter le contexte de l'appel de fonction, en prenant soin de trouver l'adresse du bloc de base contenant le branchement en question. Il n'est pas nécessaire à ce niveau d'avoir plus d'informations sur la destination de l'appel, ni les adresses des différents blocs qui composent la fonction appelée. On peut donc construire un automate correspondant au point d'appel d'une fonction en utilisant simplement l'adresse du bloc contenant ce branchement, et l'adresse en séquence correspondant au retour de la fonction.

La figure 5.2 montre un automate d'annotation supportant un contexte d'appel qui correspondrait à un point d'appel dans le bloc d'adresse `0x8424` et avec un bloc en séquence correspondant au retour de la fonction à l'adresse `0x8430`. L'utilisation de l'étoile dans le label de la transition d'entrée (`0x8424` $\rightarrow$ `*`) permet d'autoriser l'entrée dans n'importe quel point de la fonction appelée, ou même l'entrée dans différentes fonctions si la fonction appelée était initialement inconnue (si la destination du branchement était contenue dans un registre par exemple). De la même façon, l'étoile du label de l'arc retour (`*` $\rightarrow$ `0x8430`) permet de considérer l'ensemble des arcs existants dans le CFG qui ont pour destination le bloc d'adresse `0x8430`, quel que soit le bloc source.

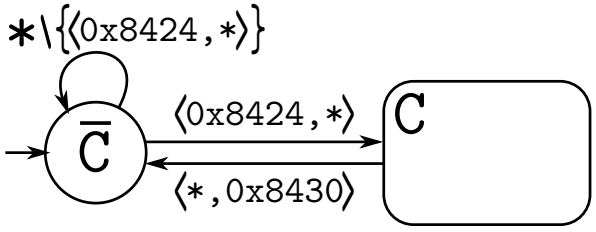


Figure 5.2 – Le contexte d'un point d'appel de fonction sous forme de HFA.

Il est possible de trouver des branchements conditionnels dans les fichiers binaires, ce qui veut dire que dans certains cas, l'appel de la fonction ne sera pas emprunté, et

l'arc `0x8424` → `0x8430` existera bel et bien dans le CFG. L'automate de la figure 5.2 ne supporte pas correctement ce type d'instructions mais en ajoutant cet arc du CFG comme transition réflexive sur l'état $\overline{\texttt{C}}$, et en utilisant les priorités des transitions, cet automate représentant un contexte et le produit qui sera fait par la suite avec le CFG resteront cohérents.

Le corps de la fonction : Le contexte représenté par le corps de la fonction correspond à l'ensemble des blocs qui appartiennent à cette fonction, incluant également les blocs des différentes fonctions qu'elle même appelle. Dans le binaire du programme, ce contexte est clairement délimité par l'adresse de début de la fonction, qui est aussi l'adresse de son premier bloc, et l'adresse de son (ou de ses) dernier(s) bloc(s). On construira donc un automate représentant le contexte du corps d'une fonction avec comme entrée un arc entre un point quelconque et le premier bloc de la fonction, et comme sortie un arc (ou des arcs) dont la source sera le dernier bloc (ou les derniers) de la fonction.

Notons par ailleurs qu'on ne traite pas le cas des fonction récursives pour lesquelles il n'existe pas de structure particulière en FFX. D'une manière générale, comme expliqué dans [51], pour que les systèmes critiques restent analysables statiquement, la récursion est interdite car à l'instar des bornes pour les boucles, il est nécessaire de connaître la profondeur de la récursion pour obtenir un WCET fini. Hors, cette profondeur maximale ne peut pas être déterminée statiquement puisqu'elle dépend de l'état des variables pendant l'exécution. Ajouté à cela, la demande en terme d'espace sur la pile pour supporter la récursion est également inconnue. Il en ressort une impossibilité d'allouer statiquement la taille maximale de la pile avant l'exécution, ce qui perturbe donc le dimensionnement du système.

La figure 5.3 montre un automate d'annotation supportant un contexte du corps d'une fonction dont le premier bloc aurait l'adresse `0x8480` et avec un bloc en séquence correspondant au retour de la fonction depuis le bloc d'adresse `0x849C`. L'utilisation de l'étoile dans le label de la transition d'entrée (`*` → `0x8480`) permet d'entrer dans ce contexte quel que soit le point d'appel de la fonction. À l'inverse, le label de la transition de sortie (`0x849C` → `*`) permet d'autoriser tous les arcs du CFG qui ont pour source le dernier bloc de la fonction, sans s'intéresser à leur bloc de destination.

Les derniers blocs d'une fonction se terminent typiquement par une instruction de branchement vers l'adresse de retour sauvegardée dans un registre spécifique ou sur la

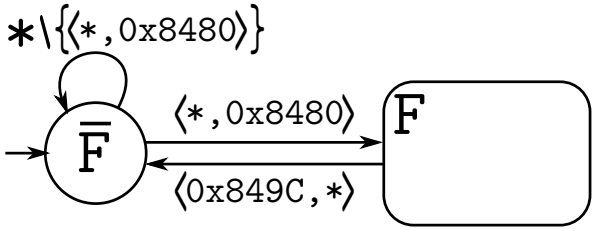


Figure 5.3 – Le contexte d'un corps de fonction sous forme de HFA.

pile. Dans les fichiers binaires compilés depuis du code source en langage C, il n'y a pas d'instruction en séquence et pas d'arc correspondant après un branchement de retour d'une fonction, mais pour d'autre types de compilation, ou pour du binaire écrit à la main, il n'est pas impossible de trouver des cas où le branchement de retour de fonction apparaît au milieu de celle-ci (une sortie au milieu d'une boucle par exemple, ou un branchement conditionnel). Dans ce cas, le bloc de retour de la fonction sera bien la source de deux arcs dans le CFG : l'un sera le branchement vers l'adresse de retour, l'autre permettra d'accéder à la suite de la fonction en séquence. Dans la pratique, on utilisera le typage des arcs fourni par Otawa pour savoir si l'arc rencontré est un arc de retour de fonction ou non.

5.3.2 Contexte de boucle

Les boucles sont des structures de contrôle qui permettent d'exécuter plusieurs fois un même ensemble d'instructions. Le contexte délimité par une boucle correspond donc à l'ensemble des exécutions des instructions qu'elle contient, depuis son point d'entrée jusqu'à sa sortie.

On considère un ensemble de blocs comme une *boucle naturelle* si un des blocs de cet ensemble domine [1] tous les autres et si ce bloc est la destination d'un arc retour. On appelle alors ce bloc la *tête de boucle* et pour le reste des blocs on parlera généralement de *corps de boucle*. Cette définition ne s'applique qu'aux boucles régulières mais il est possible de rencontrer des boucles irréductibles avec plusieurs entrées, auquel cas la tête de boucle n'est pas définie. Dans la pratique, on utilisera la gestion de ces cas particuliers par Otawa qui est capable de restructurer ces boucles non régulières. Ainsi de notre côté, on ne rencontrera que des boucles régulières dans les CFGs sur lesquels nous travaillerons.

1. Dans un CFG, un bloc `A` domine un bloc `B` si tous les chemins qui mènent du premier bloc au bloc `B` passent par `A`

Le contexte délimité par une boucle regroupe donc l'ensemble des blocs qui la composent. Ainsi, l'entrée dans ce contexte correspond à l'ensemble des arcs qui ont pour destination la tête de la boucle et qui ne sont pas des arcs retours, et la sortie du contexte est l'ensemble des arcs qui ont pour source un bloc de la boucle (de la tête ou du corps) et pour destination un bloc extérieur à la boucle.

Considérant une boucle donnée, il est possible de séparer l'ensemble des blocs d'un CFG en trois partitions que sont la *tête* de la boucle, le *corps* de la boucle, et les blocs extérieurs à la boucle (*ext*). L'automate de la figure 5.4 résume schématiquement les entrées et sorties d'un contexte de boucle en fonction de ces partitions.

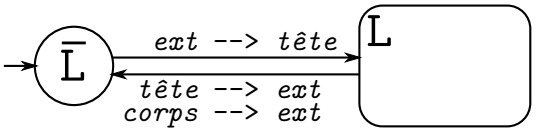


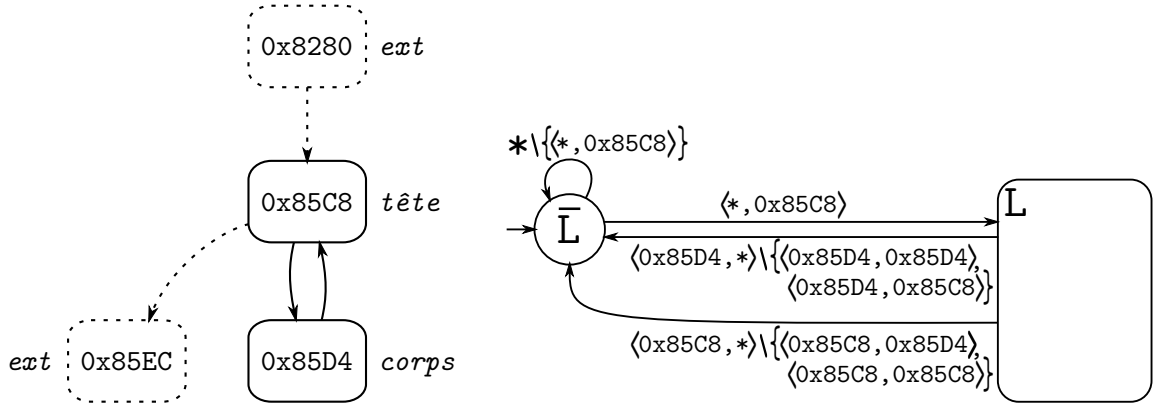
Figure 5.4 – Entrées et sorties schématiques d'un contexte de boucle sous forme d'automate hiérarchique.

En appliquant ce partitionnement à un vrai CFG, il est donc possible de construire un automate hiérarchique représentant le contexte d'une boucle tout en n'utilisant que les adresses des blocs qui la composent.

La figure 5.5a montre un exemple de CFG comportant une boucle ainsi que l'appartenance des blocs aux trois partitions citées précédemment. La figure 5.5b quant à elle représente le contexte de la boucle sous forme d'automate hiérarchique. On retrouve bien les entrées vers la tête de boucle et les sorties depuis n'importe quel point de la boucle vers l'extérieur. Notons que les adresses nécessaires pour construire cet automate sont bien celles des seuls blocs qui composent la boucle.

Dans l'étiquette de la transition d'entrée du contexte de boucle `L`, on utilisera l'étoile (`*` → `0x85C8`) pour faire la correspondance avec n'importe quel arc du CFG ayant pour destination la tête de boucle d'adresse `0x85C8`. Notons qu'on aurait pu exclure de cette transition d'entrée les différents arcs retours de la boucle (ici l'arc `0x85D4` → `0x85C8`), mais le contexte délimite justement la boucle en elle-même. Cela implique qu'il est impossible de rencontrer les arcs retours ailleurs que dans le contexte en question, ce qui, par construction, exclut la possibilité de rentrer dans ce contexte via un des arcs retour. On peut donc se permettre d'omettre l'exclusion des arcs retours dans cette étiquette d'entrée afin d'avoir un automate plus compact et lisible.

Les possibilités de sortie du contexte sont listées ici de façon exhaustive puisqu'on



(a) CFG **(b)** HFA

Figure 5.5 – Un CFG comportant une boucle (a) et le HFA représentant ce contexte de boucle (b).

autorise avec l'étoile toutes les transitions issues des blocs de la boucle en excluant celles qui restent internes à la boucle. On obtient ainsi un automate particulièrement détaillé dans lequel beaucoup d'étiquettes ne correspondent à aucun arc du CFG mais par la suite, l'opération de produit permettra de les supprimer.

En utilisant Otawa, il est possible de simplifier cet automate grâce aux outils de typage de classification des blocs des différentes boucles d'un programme. Ainsi, grâce à une pré-analyse du CFG, il est possible de ne conserver que les étiquettes correspondant à des arcs existants.

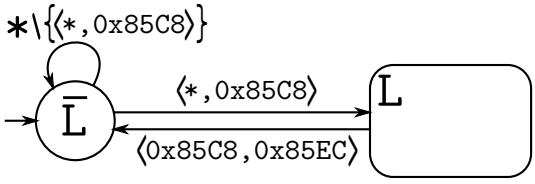


Figure 5.6 – HFA représentant un contexte de boucle après simplification des transitions sortantes.

La figure 5.6 montre une version simplifiée de l'automate de la figure 5.5b après utilisation de la classification des arcs fournie par Otawa. La transition permettant de sortir du contexte correspond désormais à un seul arc du CFG (`0x85C8` → `0x85EC`), qui faisait bien partie des transitions sortantes du contexte $\boxed{\mathtt{L}}$ dans la figure 5.5b

Notons également qu'on ne distingue pas ici les différentes itérations de la boucle, simplement la première entrée ainsi que la sortie, mais il est possible d'utiliser les mêmes partitions pour exprimer le contexte correspondant à une itération.

5.3.3 Contexte d'itération

Le comportement d'un programme d'une itération à l'autre peut changer, parce que le flot de contrôle dépend de l'incrément de la boucle par exemple, parce que des instructions d'une itérations ont un impact sur les suivantes, ou à cause de variables globales. Par ailleurs, lorsqu'un expert veut exprimer une propriété relative à chaque itération d'une boucle, il peut l'exprimer relativement à la borne de la boucle et ainsi la rendre globale, mais il peut également vouloir l'exprimer localement et la répéter pour toutes les itérations. Il est donc intéressant d'être capable de délimiter le contexte correspondant à chaque itération séparément pour fournir une liberté accrue dans l'expression de propriétés locales.

Alors que le contexte d'une boucle (naturelle) est délimité par son arc d'entrée et ses arcs de sortie, le contexte correspondant à une itération est délimitée par les arcs qui partent de la tête de boucle vers le corps de la boucle et ceux qui reviennent du corps de la boucle vers la tête, (ou vers l'extérieur pour la dernière itération).

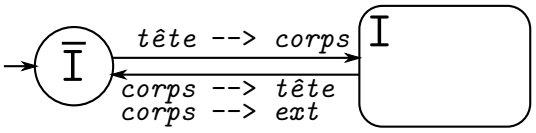


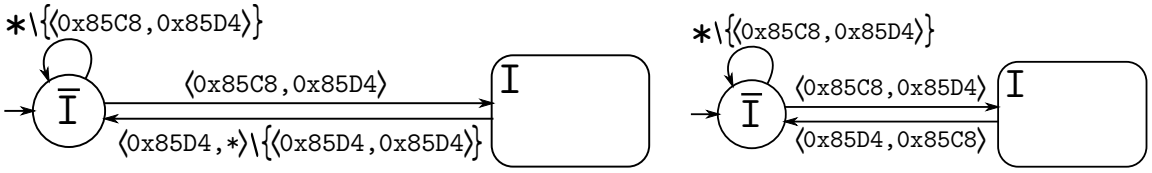
Figure 5.7 – Entrées et sorties schématiques d'un contexte d'itération sous forme d'automate hiérarchique.

En utilisant les trois partitions définies dans la partie 5.3.2, la figure 5.7 illustre schématiquement sous forme de HFA les entrées et sorties d'un contexte correspondant à une itération de boucle.

Notons que la sortie d'une itération peut se faire par deux types d'arcs : les arcs qui vont du *corps* vers la *tête* de boucle, qui correspondent donc à ses différents *arcs retours*, mais également par les arcs qui permettent de sortir de la boucle directement depuis son *corps*. En effet, lors de la dernière itération de la boucle, si la sortie de la boucle ne se fait pas depuis la *tête* de boucle, l'arc emprunté ira bien du *corps* vers l'*extérieur* de la boucle.

L'exemple de contexte d'itération présenté dans la figure 5.8a se base sur la boucle présente dans le CFG de la figure 5.5a. L'entrée dans l'itération (`0x85C8` $\rightarrow$ `0x85D4`) correspond bien à l'arc qui va de la *tête* au *corps* de la boucle. La sortie est quant à elle exprimée grâce à l'étoile (`0x85D4` $\rightarrow$ `*`) pour autoriser tous les arcs qui ont pour source le corps de la boucle, à l'exception des arcs qui restent dans le corps de la boucle.

Comme il n'y a qu'un seul bloc dans le corps de la boucle, le seul arc à exclure est l'arc `0x85D4` → `0x85D4`. On capture ainsi les blocs qui vont du *corps* vers la *tête* mais aussi ceux qui vont du *corps* vers l'*extérieur* de la boucle.



(a) HFA **(b)** HFA simplifié

Figure 5.8 – Un HFA représentant un contexte d'itération de boucle (a) et le même HFA simplifié (b).

À l'instar de la simplification illustrée sur la figure 5.6, il est possible de faire une pré-analyse du CFG par Otawa afin de rendre plus compact l'automate représentant le contexte d'une itération. La figure 5.8b montre une version simplifiée du HFA de la figure 5.8a. En analysant le CFG, on se rend compte qu'il n'existe pas d'arc entre le *corps* et l'*extérieur* de la boucle, ni d'arc réflexif sur le bloc `0x85D4`. Le seul arc permettant de sortir du contexte de l'itération est donc l'arc retour `0x85D4` → `0x85C8`.

5.4 Du FFX aux automates

Nous avons montré dans la partie 5.3 comment il est possible de fabriquer des automates hiérarchiques représentant les différents contextes qu'on rencontre régulièrement dans un programme. En utilisant ces constructions et les possibilités offertes par le formalisme dans le support de contraintes numériques, il est désormais possible de traduire des annotations issues d'un langage d'annotation comme FFX en HFAs.

Dans cette partie, nous appliquerons donc cette traduction à FFX pour montrer comment les annotations exprimées dans ce langage peuvent être traduites efficacement en automates hiérarchiques à contraintes.

Annotation de source ou de binaire

Les diverses annotations supportées par FFX comportent une information de localisation, sous forme de ligne dans un fichier de code source, d'adresse physique relative au fichier binaire du programme analysé, ou encore de label correspondant au nom d'un élément comme un nom de fonction. Comme expliqué dans la partie 5.2,

lorsque les informations de localisation correspondent à une ligne dans un fichier source, ou à un label, on utilisera les informations de débogage associées à la compilation pour retrouver les adresses physiques correspondantes dans le fichier binaire. En cas d'échec de cette mise en correspondance (à cause d'optimisations par exemple), il sera évidemment impossible de fabriquer un automate hiérarchique correspondant au FFX et donc d'intégrer l'annotation en question à l'analyse de WCET. Notons cependant que cette limitation n'est pas liée à l'utilisation des automates d'annotations mais s'applique à l'analyse dans tous les cas.

5.4.1 Traduction des appels de fonction

Le format FFX fournit deux balises qui permettent de gérer les appels de fonction :

- La balise <`call`> (code 5.4) qui correspond typiquement au contexte d'un point d'appel de fonction présenté dans la partie 5.3.1 à la page 76.
- La balise <`function`> (code 5.5) qui correspond au contexte du corps d'une fonction présenté dans la partie 5.3.1 à la page 78.

```
CALL ::=
<call LOCATION-ATTRS INFORMATION-ATTRS>
| FUNCTION+
</call>
```

Code 5.4 – Balise FFX d'un appel de fonction.

```
FUNCTION ::=
<function LOCATION-ATTRS INFORMATION-ATTRS>
| STATEMENT*
</function>
```

Code 5.5 – Balise FFX d'une fonction.

Dans ces deux balises, on trouve notamment l'information de localisation, comme une adresse physique, une paire ⟨*fichier source*, *numéro de ligne*⟩ ou un *label*. Une partie de la grammaire FFX est fournie en appendice A.

Pour un appel de fonction donné, on trouve habituellement ces deux balises imbriquées. Le code 5.6 montre un appel à la fonction *foo* dans le programme principal *main* à la ligne 17. La figure 5.9 représente le CFG de cet appel de fonction.

Il est possible de générer un fichier FFX comportant des informations de flot sur ce code source, en utilisant un outil comme oRange, ou même d'écrire ce fichier à la main. Le code 5.7 est issu d'une génération automatique et représente l'appel ainsi que la fonction *foo* en elle-même. On trouve pour la balise <`call`> des informations

```
void foo(){
  ...
}
int main(){
  foo(); // ligne 17
}
```

Code 5.6 – Exemple de code source avec un appel à une fonction *foo*.

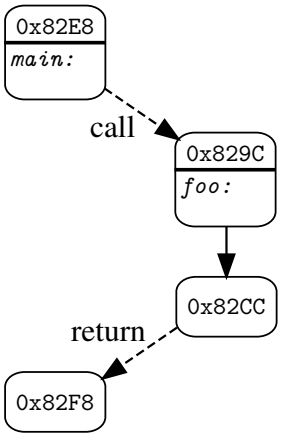


Figure 5.9 – CFG illustrant l'appel à *foo*.

de localisation issues du code source, ainsi que le nom de la fonction, tandis qu'on ne trouve que le label de la fonction dans la balise <`function`>. Il faut donc utiliser les informations de débogage pour connaître les adresses physiques correspondantes à chaque élément et construire le HFA de la figure 5.10 comme décrit dans la partie 5.3.1.

```
<call source="main.c" line="17">
  <function name="foo">
  </function>
</call>
```

Code 5.7 – Fichier FFX généré à partir du code 5.6.

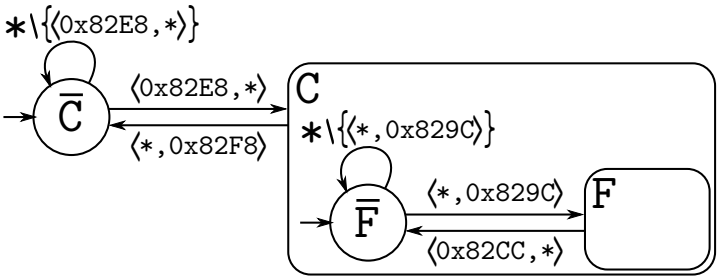
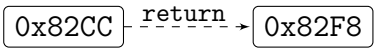
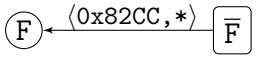


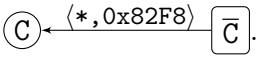
Figure 5.10 – FFA représentant les contextes des balises imbriquées `call` et `function` du code 5.7

Dans cet automate, l'imbrication des contextes qu'on trouve dans le code FFX est restituée au moyen de la hiérarchie d'automates. Le super-état `C` correspond au contexte de la balise <`call`>, et contient naturellement le sous-automate permettant l'entrée dans le contexte de la balise <`function`>.

En mettant en parallèle le CFG de la figure 5.9 avec cet automate, on peut voir que l'arc `0x82E8` -`call`-> `0x829C` permet de rentrer à la fois dans le contexte `C` via C̄ -⟨0x82E8,∗⟩-> `C` mais également dans le contexte `F` via F̄ -⟨∗,0x829C⟩-> `F`. Ainsi, cette entrée dans les deux contextes se fait bien de façon simultanée avec un seul et même arc du CFG.

De la même façon, l'arc 

permet d'emprunter simultanément les transitions de sorties 

et 

.

5.4.2 Traductions des boucles

Le format FFX supporte les annotations de bornes de boucles d'un programme. Pour cela, il existe une balise <loop> permettant d'identifier une boucle par ses information de localisation, à l'instar des balises <call> et <function>, et qu'il est possible de borner en renseignant des attributs `maxcount` et `totalcount`. Par ailleurs, le format supporte également les précisions concernant une itération spécifique, par l'utilisation de la balise <iteration> accompagnée de son numéro en attribut.

Le code 5.8 montre la grammaire de ces deux balises. Notons que quand une itération est précisée, elle est toujours imbriquée dans la balise de la boucle correspondante.

```
LOOP ::=
  <loop LOCATION-ATTRS INFORMATION-ATTRS> STATEMENT* </loop>
| <loop LOCATION-ATTRS INFORMATION-ATTRS>
    <iteration number="INT"> STATEMENT* </iteration>
  </loop>
```

Code 5.8 – Balise FFX d'une boucle avec et sans la balise optionnelle correspondant à une itération de cette boucle

5.4.2.1 Les bornes de boucles et les contextes

Les deux attributs `maxcount` et `totalcount` que l'on trouve dans les balises <loop> permettant de borner la boucle ont des sens différents : le premier indique le nombre d'itérations maximal à chaque fois qu'on entre dans la boucle, alors que le second donne le nombre total d'exécutions de cette boucle dans le programme. Ces bornes ne couvrent que les apparitions de la boucle dans un contexte précis, correspondant au contexte englobant la balise <loop> dans le fichier FFX.

Le code 5.9 montre un exemple où apparaît une boucle dans une fonction `foo`, et où cette fonction est appelée depuis le *main* mais également depuis une seconde fonction `bar`. Le code 5.10 montre diverses annotations pour la boucle de la fonction `foo`. Dans le contexte de la fonction `bar`, le nombre maximal d'itérations est 10, tandis que le

```
void foo(int n){
  for(int i=0; i<n; i++);
}
void bar(){
  foo(10);
}
int main(){
  bar();
  bar();
  foo(50); // ligne 19
}
```

Code 5.9 – Exemple de boucle et de multiples appels de fonctions.

```
<function name="bar">
  <loop maxcount="10" totalcount="20">
  </loop>
</function>
<call name="foo" line="19">
  <loop maxcount="50" totalcount="50">
  </loop>
</call>
<function name="foo">
  <loop maxcount="50" totalcount="70">
  </loop>
</function>
```

Code 5.10 – Différentes valeurs des bornes d'une même boucle en fonction du contexte englobant.

total est de 20 puisque la fonction principale appelle deux fois `bar`. Cette première annotation concerne donc bien uniquement les apparitions de la boucle dans le contexte `bar`. La seconde annotation concerne les apparitions de la boucle lorsque la fonction `foo` est appelée depuis la ligne 19 du fichier source, c'est à dire directement depuis le `main`. Dans ce contexte, le nombre total et le nombre maximum d'itérations sont les mêmes puisqu'on ne s'intéresse qu'à un seul appel à la fonction `foo`. Enfin, la troisième annotation dépend du contexte de la fonction `foo`, or la boucle apparaît toujours dans cette fonction, donc le `maxcount` correspond au pire `maxcount` rencontré tandis que le total correspond à la somme de toutes les apparitions de la boucle.

5.4.2.2 L'expression des bornes par les HFAs

Nous nous intéressons dans cette partie à la caractérisation de la boucle et de sa borne, au moyen d'un HFA. La traduction, pour les entrées et sorties du contexte de boucle, des informations de localisation (`LOCATION-ATTRS`) en adresses physiques correspondantes a été couverte dans la partie 5.3.2 et le fonctionnement rejoint celui de la partie 5.4.1. Pour cette raison, nous utiliserons le CFG de la figure 5.11 pour raisonner sur les arcs d'entrée, de sortie et les différentes itérations d'une boucle.

Les codes 5.11 et 5.12 montrent l'expression des bornes de cette boucle au format FFX, et le second code précise également un contexte d'itération dont le numéro "`*`" signifie *pour toute itération*. Ce contexte peut sembler superflu mais il permet en réalité d'exprimer des contraintes localement à une itération, et de répéter cette contrainte locale pour chaque itération de la boucle. Ce n'est pas directement équivalent à expri-

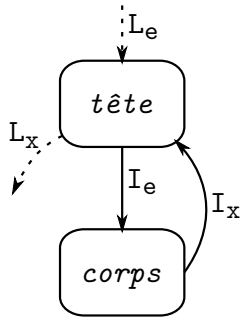


Figure 5.11 – CFG simplifié d'une boucle

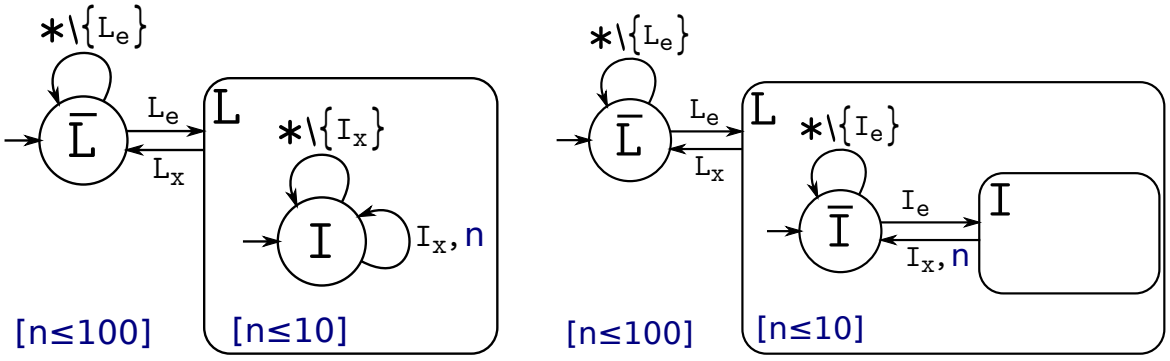
```
<loop maxcount="10" totalcount="100">
</loop>
```

Code 5.11 – FFX avec bornes

```
<loop maxcount="10" totalcount="100">
  <iteration number="*">
  </iteration>
</loop>
```

Code 5.12 – FFX avec bornes et contexte d'itération

mer cette même contrainte sous forme globale dans le contexte de la boucle mais elle peut néanmoins être obtenue en suivant le principe de la partie 4.4.3.3. Nous verrons des exemples d'utilisation de ce contexte dans le chapitre 6.



(a) FFA avec bornes

(b) FFA avec bornes et contexte d'itération

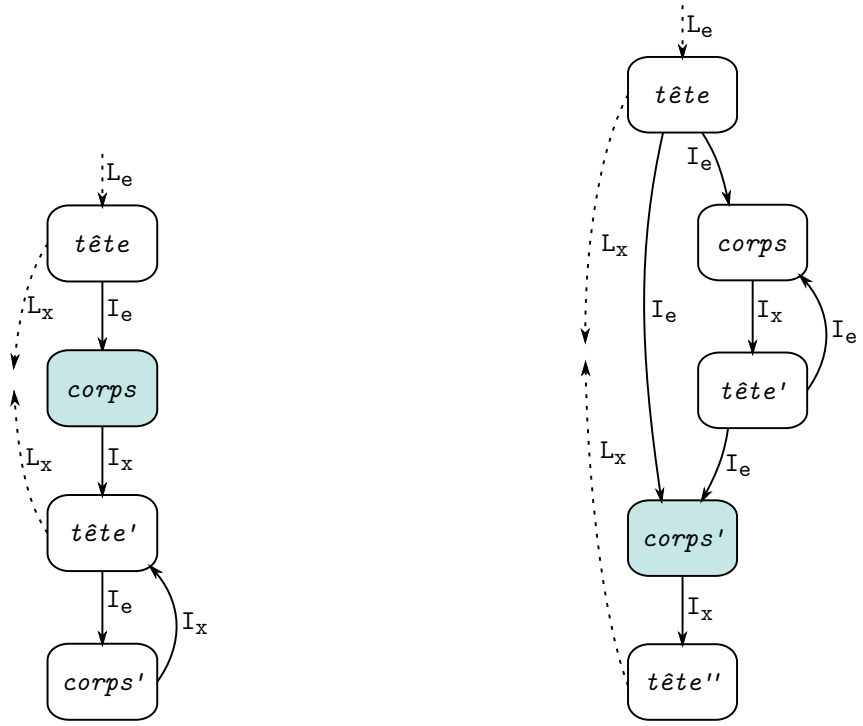
Figure 5.12 – FFA **a** (resp. **b**) généré à partir du code 5.11 (resp. 5.12)

Les FFAs de la figure 5.12 supportent les annotations de bornes de boucle issues des codes 5.11 et 5.12. En attachant une variable `n` à l'arc retour $\texttt{I}_\texttt{x}$, on a pu exprimer le `maxcount` localement avec la contrainte [n≤10] dans le contexte de la boucle, et le `totalcount` avec la contrainte [n≤100] à la racine de l'automate. La contrainte locale (le `maxcount`) sera ainsi adaptée lors de l'opération d'aplatissement suivant le processus décrit dans la partie 4.4.3.3 alors que ça ne sera pas le cas pour la contrainte correspondant au `totalcount` puisque celle-ci est déjà globale.

5.4.2.3 Déplier une itération précise

Lorsqu'on s'intéresse de près à une boucle, on souhaite parfois isoler une itération précise, afin de fournir des informations supplémentaires sur le flot de contrôle dans

celle-ci. Par exemple, il n'est pas rare de rencontrer des boucles comportant une phase d'initialisation, qui présentent donc un code activé uniquement lors de la première itération, ou un comportement spécifique pour la dernière itération. La figure 5.13 montre des dépliages possibles d'une itération spécifique de la boucle de la figure 5.11.



(a) Première itération **(b)** Dernière itération

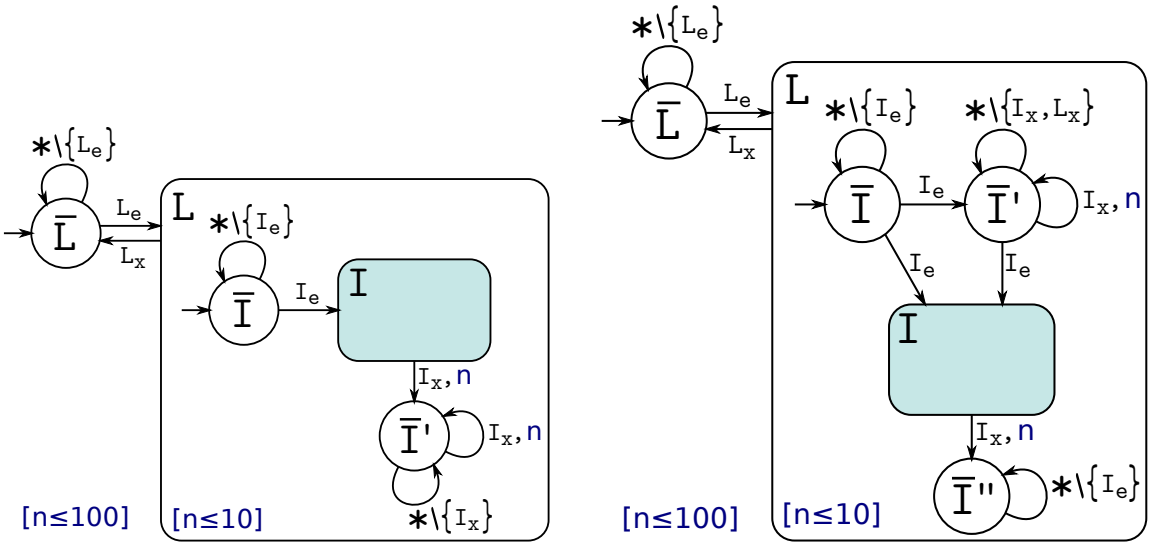
Figure 5.13 – Dépliages partiels d'une boucle avec isolation d'une itération précise.

Dans le premier dépliage (graphe 5.13a), on a isolé la première itération et dupliqué pour cela la tête et le corps de la boucle. Ainsi, quel que soit le nombre d'exécutions de la boucle, ce dépliage permettra d'isoler la première itération. De plus, ce graphe reste déterministe et la borne du nombre d'itérations s'appliquera sur la somme des arcs retours $\mathtt{I_x}$.

Le graphe 5.13b illustre le dépliage de la dernière itération de la boucle. En d'autres termes, on souhaite que la dernière itération exécutée soit toujours isolée, sans avoir connaissance de la borne de boucle exacte. Il est nécessaire d'introduire de l'indéterminisme ici car au moment de la construction de ce graphe, on ne sait pas réellement quelle va être la dernière itération, à moins d'avoir la borne exacte de la boucle. Ainsi, la dernière itération peut être la première, et dans ce cas il faut immédiatement activer l'itération isolée, ou bien une des suivantes, auquel cas on ne sait pas combien de fois on bouclera avant d'activer cette dernière itération. D'autre part, la tête de

boucle a été dupliquée deux fois dans ce graphe : en effet, la première tête permet d'entrer dans la boucle ou d'en sortir immédiatement dans le cas où la boucle itère zéro fois, et la dernière tête permet uniquement de sortir de la boucle puisque le corps de l'itération isolée vient d'être activé. Reste ainsi la seconde tête qui a la particularité de ne pas autoriser la sortie de boucle, car la dernière itération n'as pas encore été activée. Ce dépliage qui repose sur de l'indéterminisme permet donc bien d'isoler la dernière itération d'une boucle et de forcer son activation, et ce quel que soit le nombre d'itérations exact de la boucle tant que celui-ci est strictement positif.

Le langage d'annotation FFX motive ce genre de dépliage par l'existence de l'attribut "`number`" dans la balise <`itération`> qui peut contenir une valeur entière (1 pour la première et -1 pour la dernière notamment). Il est possible de construire des automates spécifiques permettant d'isoler la première ou la dernière itération d'une boucle précise dans le but d'aboutir aux dépliages illustrés sur la figure 5.13.



(a) Première itération **(b)** Dernière itération

Figure 5.14 – Automate **a** (resp. **b**) permettant d'isoler le contexte de la première (resp. dernière) itération.

L'automate 5.14a permet de déplier la première itération d'une boucle. Il pourra être construit à partir d'une annotation FFX similaire au code 5.12 comportant un contexte d'itération avec l'attribut `number="1"`. La transition $(\overline{\mathtt{I}}) \xrightarrow{\mathtt{I_e}} [\mathtt{I}]$ du sous-automate capture l'entrée dans la première itération et isole son contexte. La sortie de ce contexte $[\mathtt{I}] \xrightarrow{\mathtt{I_x}} (\overline{\mathtt{I}}')$ aboutit dans un troisième état qui permettra de capturer le reste des itérations de la boucle. Le décompte des itérations se fait toujours sur la variable `n` qui est associée aux deux apparitions de l'arc retour $\mathtt{I_x}$ et au moyen d'une

contrainte locale pour le `maxcount` et une globale pour le `totalcount`.

L'automate 5.14b permet d'isoler la dernière itération d'une boucle et devrait être construit lorsque la balise <`iteration`> d'une boucle comportera un attribut `number="-1"`. Cependant il contient de l'indéterminisme qui n'est pas supporté par le formalisme actuel des HFAs, mais qui reste nécessaire puisque le résultat recherché est lui-même un CFG indéterministe, comme le montre le graphe 5.13b. Nous souhaitons ici donner l'intuition qu'il est possible de construire un HFA indéterministe qui permettrait d'aboutir au résultat voulu.

Dans le sous-automate de $\boxed{\texttt{L}}$, l'entrée de la première itération pourra être capturée par la transition $\overline{\texttt{I}} \xrightarrow{\texttt{I}_\texttt{e}} \overline{\texttt{I}}'$, mais également par l'entrée dans le contexte isolé de la dernière itération via $\overline{\texttt{I}} \xrightarrow{\texttt{I}_\texttt{e}} \boxed{\texttt{I}}$, ce qui correspond au cas ou la première itération est aussi la dernière. Lorsqu'on se trouve dans l'état $\overline{\texttt{I}}'$, les tours sont décomptés grâce à la variable `n` sur l'arc retour, mais la sortie de la boucle via $\texttt{L}_\texttt{x}$ et interdite, pour forcer le passage dans le contexte $\boxed{\texttt{I}}$ lors de la dernière itération. La sortie du contexte de la dernière itération aboutira pour finir dans l'état $\overline{\texttt{I}}''$ où une nouvelle entrée dans une itération via $\texttt{I}_\texttt{e}$ sera interdite.

5.5 Les contraintes numériques

Un des éléments clé dans la précision de l'estimation du WCET d'un programme est le nombre de chemins infaisables tirés des annotations qu'il a été possible d'intégrer dans l'analyse. Les bornes de boucles par exemple sont nécessaire pour obtenir un premier résultat mais il est possible de fournir des informations additionnelles susceptibles d'éliminer de chemins infaisables supplémentaires, au moyen par exemple de contraintes numériques. Par exemple, un expert ou une analyse automatique sera parfois capable d'exprimer une exclusivité entre deux arcs du CFG, ou entre deux conditions du programme source, ce qui pourrait permettre, à terme, de réduire le pessimisme de l'analyse de WCET.

Le format FFX supporte ce genre de contraintes numériques grâce à la balise <`control-constraint`> associée à des éléments déclarés comme des identifiants. La grammaire de cette balise est fournie en appendice A.

Habituellement, ces contraintes numériques sont directement intégrées dans le système ILP utilisé dans la méthode IPET par Otawa. Les HFAs peuvent servir de support

dans le transfert des contraintes depuis le langage d'annotation vers ce système ILP, mais ils offrent également la possibilité de manipuler ces contraintes pour les représenter sous forme d'automate déplié, et les intégrer ainsi non plus comme contraintes numériques mais dans la structure même du CFG. Nous ne détaillerons pas ce point ici car il sera au cœur du chapitre 6. Nous y verrons notamment que l'intégration sous forme de dépliage du CFG permet d'améliorer la précision des analyses bas niveau. En effet, la duplication de certains blocs réduit parfois le nombre de chemins possibles dans le CFG et supprime par la même occasion certaines opérations d'unions (sur les états des caches abstraits principalement) qui étaient responsables de l'injection de pessimisme dans les analyses sur la modélisation du matériel.

5.6 Conclusion

Nous avons montré dans ce chapitre comment il était possible de construire des HFAs à partir des annotations de flot exprimées dans le langage FFX et utilisées habituellement dans l'estimation de WCET par la méthode IPET. Les contextes des différentes structures d'un programme, à savoir les appels de fonction, les fonctions, les boucles et les itérations de boucles ont été représentés sous forme de HFAs dans lesquels on a montré comment il était possible d'exprimer des attributs comme les bornes de boucles locales et totales.

Par ailleurs, l'étoile ($*$) permettant de représenter des ensembles de transitions a été introduite ainsi que des règles de priorité destinées à condenser et simplifier la représentations des HFAs.

Enfin, des structures particulières destinées à déplier des contextes spécifiques comme une itération précise d'une boucle ont été présentées, et nous avons donné l'intuition du dépliage de CFG qu'il était possible de faire en représentant des contraintes numériques sous forme d'automates, ainsi que les bénéfices potentiels qui pouvaient en découler, mais ce point fera l'objet du chapitre suivant.

"Beneath this mask there is more than flesh. Beneath this mask there is an idea, Mr. Creedy, and ideas are bulletproof."

— V

6

Compromis entre dépliage du CFG et intégration par contraintes

Sommaire

6.1 Introduction

Dans la partie 2.5.2, nous mettons en avant la possibilité offerte par l'expressivité des automates pour intégrer des annotations de flot dans la structure même du CFG plutôt qu'au moyen d'une contrainte ILP supplémentaire. On détaille dans ce chapitre

ces deux méthodes d'intégration et leurs impacts respectifs sur l'estimation de WCET par la méthode IPET. Ce chapitre reprend et étend l'article [46].

Dans un premier temps, nous illustrerons le dépliage partiel de CFG sur un exemple, afin de mettre en évidence les différences entre le résultat de l'intégration d'annotation par ce dépliage et l'intégration par l'ajout de contraintes ILP.

On présentera ensuite des expérimentations permettant de comparer les résultats issus de ces deux méthodes d'intégration. On présentera notamment les modules du *plug-in* intégré dans l'outil Otawa et développé spécifiquement pour gérer la transformation de fichiers FFX en automates et pour intégrer les annotations portées par ces automates dans une analyse de WCET par IPET.

Enfin, nous montrerons comment il est possible d'exprimer systématiquement des contraintes numériques sous forme de restrictions structurelles des automates, et comment ce procédé peut aboutir à un dépliage partiel du CFG et à des bénéfices de précision de l'estimation de WCET. Pour cela, nous utiliserons des résultats expérimentaux issus d'un programme spécifique destiné à faire la preuve de concept de cette approche.

6.2 Un premier exemple d'intégration par dépliage

On présente dans cette partie un exemple de programme comportant un chemin infaisable dans chaque itération d'une boucle. On illustre notamment l'intégration de cette annotation par des HFAs dans le CFG du programme, en utilisant les opérations présentées dans le chapitre 4.

6.2.1 Le programme

L'exemple utilisé pour illustrer le dépliage partiel de CFG est présenté dans le code 6.1. Ce programme comporte une boucle dans laquelle, pour chaque itération, les conditions `A` et `B` sont exclusives.

La figure 6.1 présente le CFA qui peut être généré à partir du CFG de ce programme. On a mis en avant les arcs de ce graphe qui nous intéresseront dans la suite, à savoir les arcs permettant d'entrer et de sortir d'une itération (respectivement `E` et `X`) et les arcs correspondants aux conditions exclusives (`A` et `B`).

```
int main() {
    for(int i = 0; i < 11*13; i++) {
        if(i % 11 == 0) {
            ... // A
        }
        if(i % 13 == 0) {
            ... // B
        }
    }
}
```

Code 6.1 – Programme présentant deux conditions mutuellement exclusives dans chaque itération d'une boucle.

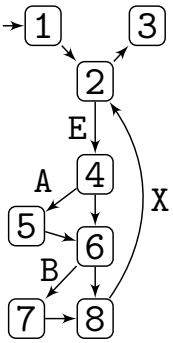


Figure 6.1 – CFG du programme 6.1.

Le WCET estimé par la méthode IPET suppose de maximiser la fonction objectif en considérant un chemin d'exécution qui passerait dans `A` et `B` pour chaque itération de la boucle, or il apparaît clairement que ce chemin est infaisable en réalité et que la durée correspondante constitue une borne bien supérieure au WCET réel du programme. L'inclusion du chemin infaisable indiquant l'exclusivité entre les conditions `A` et `B` permettra donc de réduire la sur-estimation du WCET par la méthode IPET.

6.2.2 L'annotation en FFX et sous forme de HFA

Il est possible d'exprimer l'exclusivité entre les branches `A` et `B` mentionnée dans la partie précédente en FFX, grâce à la balise <`control-constraint`> qui supporte des contraintes numériques associées à des éléments du CFG.

Le code 6.2 montre un fichier FFX exprimant une exclusivité entre les arcs $4 \xrightarrow{A} 5$ et $6 \xrightarrow{B} 7$ pour chaque itération de la boucle (<`iteration number="*"`>). La balise <`control-constraint`> exprime pour cela la contrainte numérique $x_A + x_B \leq 1$.

Il existe deux possibilités pour représenter cette annotation sous forme de HFA.

La première consiste à utiliser un compteur associé à une contrainte locale au contexte d'itération. Cette solution est illustrée par l'automate de la figure 6.2. On a représenté par un super-état le contexte partiel [1] de l'itération de la boucle dans lequel l'exclusion entre les deux arc `A` et `B` est définie, et on a attaché un compteur α à la transition factorisée correspondant à ces deux arcs. La contrainte $[\alpha \leq 1]$ a ainsi pu être définie dans le sous-automate du contexte d'itération de façon à ce qu'elle soit

1. Les contextes de la boucle et de la fonction `main` ont été omis pour simplifier la représentation.

```
<function name="main">
  <loop source="excl.c" line="2">
    <iteration number="*">
      <control-constraint>
        <le>
          <add>
            <count src="BB_4" dst="BB_5"/>
            <count src="BB_6" dst="BB_7" />
          </add>
          <int>1</int>
        </le>
      </control-constraint>
    </iteration>
  </loop>
</function>
```

Code 6.2 – Fichier FFX supportant une annotation d'exclusion sous forme de contrainte numérique.

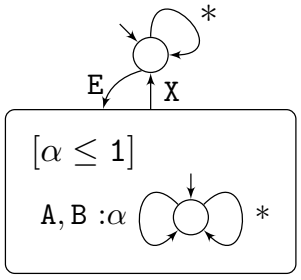


Figure 6.2 – HFA avec contrainte.

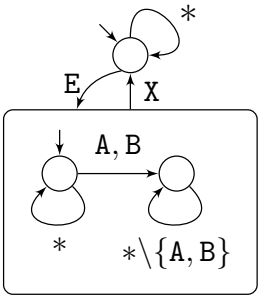


Figure 6.3 – HFA déplié.

vérifiée à chaque sortie de ce contexte comme expliqué dans le chapitre 4.

La seconde possibilité est d'utiliser un automate déplié, intégrant l'exclusivité dans sa structure. L'automate de la figure 6.3 illustre cette solution. Le contexte partiel de l'itération est le même que celui de la figure 6.2 mais le sous-automate correspondant à l'exclusion est cette fois composé de deux états. Le premier état accepte toutes les transitions de l'alphabet, mais les labels `A` ou `B` permettent de transiter dans un état qui interdit ces deux arcs spécifiques. Ainsi, ce sous-automate interdit les mots qui comportent à la fois un `A` et un `B` (ainsi que les mots qui comportent plusieurs `A` ou plusieurs `B`, mais par construction, ces mots ne peuvent pas apparaître dans une même itération).

6.2.3 Le produit avec le CFA

Nous avons présenté dans la partie précédente deux HFAs capables de représenter une annotation de flot locale pour chaque itération d'une boucle. Afin d'intégrer ces annotations dans l'estimation de WCET, il convient de suivre la méthode fournie dans le chapitre 4 qui consiste à faire une injection du CFA du programme analysé dans les

HFAs porteurs d'annotations. Les résultats de ces opérations sont présentés dans les figures 6.4 et 6.5.

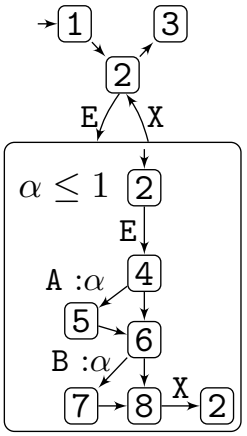


Figure 6.4 – Injection du CFA de la figure 6.1 dans le HFA de la figure 6.2.

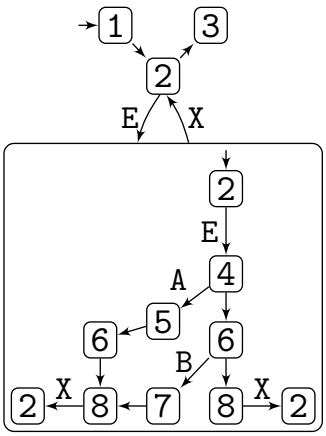


Figure 6.5 – Injection du CFA de la figure 6.1 dans le HFA de la figure 6.3.

Dans le HFA de la figure 6.4, on retrouve la structure du CFA initial à l'intérieur de l'itération, enrichie de la variable et de la contrainte du HFA utilisé. La variable α a ainsi été distribuée sur les transitions `A` et `B` et la contrainte $[\alpha \leq 1]$ a été associée au sous-automate en question.

Le HFA de la figure 6.5 présente des différence notoires puisque la structure du CFA n'a pas été conservée pendant cette opération d'injection. En effet, le sous-automate du HFA a permis de déplier le CFA lors de l'opération, en dupliquant les blocs 6, 8 et 2, de façon à ce qu'il n'existe pas de chemin dans ce graphe qui puisse passer par `A` et par `B`. La contrainte a donc bien été retranscrite dans la structure même de ce sous-automate, sans qu'aucune contrainte ni variable n'ait été utilisée.

6.2.4 L'aplatissement des HFAs

Afin de retrouver un graphe plat et d'être capable de reconstruire un CFG à partir de ces HFAs, il est nécessaire d'utiliser l'opération d'aplatissement présentée dans le chapitre 4. Les résultats de ces aplatissements sont présentés dans les figures 6.6 et 6.7.

L'aplatissement du HFA de la figure 6.4 a impliqué l'ajout d'une nouvelle variable (β) correspondant aux entrées dans chaque itération de la boucle. Cette variable a ensuite été utilisée pour adapter les contraintes locales en contraintes globales et la contrainte $[\alpha \leq 1]$ est ainsi devenue $[\alpha \leq \beta]$. Lors de la reconstruction du CFG à partir

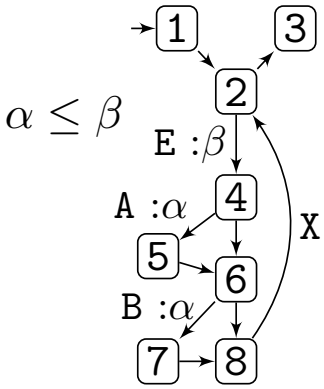


Figure 6.6 – Aplatissement du HFA de la figure 6.4.

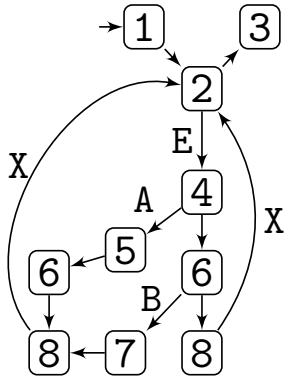


Figure 6.7 – Aplatissement du HFA de la figure 6.5.

de cet automate, cette contrainte sera ainsi traduite en la contrainte ILP suivante : $x_A + x_B \leq x_E$.

Dans la figure 6.5, l'aplatissement a simplement fait apparaître deux arcs retours au lieu de l'unique arc retour initial, et la séparation des chemins a bien été conservée pendant cette opération. L'exclusivité entre les arcs `A` et `B` a donc bien été intégrée à la structure même du CFG.

6.3 Expériences

Les deux approches différentes destinées à intégrer des annotations de flot par des HFAs méritent une comparaison en terme de précision du WCET estimé. La première approche, traditionnellement utilisée, consiste à rajouter des contraintes au système ILP. Les contraintes ainsi rajoutées sont cependant forcément globales puisque la résolution du système ILP dans la méthode IPET se fait de façon globale. Il est donc probable que cette méthode s'accompagne d'une perte de précision due à la transformation des contraintes locales en contraintes globales. La seconde approche, consistant à faire apparaître explicitement les chemins autorisés dans la structure du CFG, ne souffre pas de cette sur-estimation puisque les annotations conservent leur localité. On peut ainsi espérer un gain de précision dans l'estimation du WCET par rapport à la première approche.

6.3.1 Les outils d'analyse

Les automates et opérations présentés dans le chapitre 4 ont été implémentés sous forme de plug-in dans l'outil d'analyse statique Otawa [4].

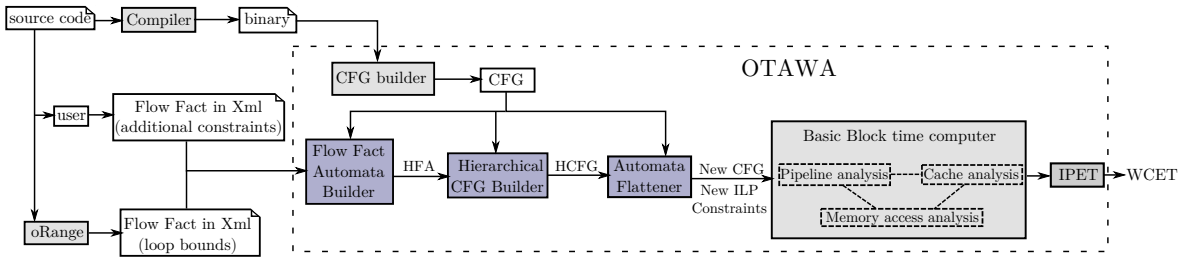


Figure 6.8 – Vue générale de notre système d'analyse de WCET comprenant l'implémentation des automates d'annotation de flot.

La figure 6.8 montre l'ensemble de la chaîne logicielle permettant d'intégrer des annotations de flot dans une analyse de WCET au moyen des automates d'annotations. Sur la partie gauche de ce schéma, on trouve le code source qui peut être annoté par un expert et analysé par l'outil oRange [44] capable de générer un fichier FFX en sortie contenant notamment les bornes des boucles du programme. Ce code source est compilé en un fichier binaire fourni à Otawa qui en construit le CFG.

Ce CFG et les annotations en FFX fournies par oRange et par l'utilisateur servent de point de départ au plug-in qui se décompose en trois modules. Le *Flow Fact Automata Builder* est chargé de *parser* le fichier FFX et de construire le HFA correspondant. Ce HFA est ensuite transféré au *Hierarchical CFG Builder* qui construit le CFA à partir du CFG du programme et l'injecte dans le HFA reçu du module précédent. Le résultat produit par ce module est un *Hierarchical CFG* (HCFG) qui doit subir une opération d'aplatissement avant d'être rendu à Otawa pour que celui-ci procède à l'analyse de WCET à proprement parler. Cet aplatissement est pris en charge par le troisième module *Automata Flattener* qui, le cas échéant, intègre également les nouvelles contraintes dans le système ILP.

Le modèle d'architecture

Lors de cette session d'expériences, nous avons utilisé un modèle d'architecture d'Otawa dérivé du processeur ARM9 avec un cache d'instruction de 1 kilo-octet et un cache de données de 16 kilo-octets. Le cache d'instruction est un cache associatif à deux voies (*two-ways associative*) avec une politique de remplacement de type LRU (*Least*

Recently Used).

6.3.2 Analyse de programmes du WTC14

Les résultats présentés dans cette partie sont issus de l'analyse de deux programmes [2] provenant du *WCET Tool Challenge 2014* (WTC14) [1]. Le premier est le programme *runFlightPlan* qui provient de la suite de programmes *heli* qui contrôle un modèle d'hélicoptère. Le second, *tcas-a*, est tiré de la suite de programmes *tcas* qui est un logiciel de gestion de trafic. Pour ces deux programmes, nous avons pu déceler la présence de branches exclusives dans des fonction spécifiques.

Table 6.1 – Résultats de l'analyse de WCET du WTC14 avec et sans automates.

Programme	WCET avec boucles bornées	contraintes additionnelles		dépliage	
		WCET	Gain	WCET	Gain
runFlightPlan	1715	1534	10.6%	1534	10.6%
tcas-a	19880	17484	12.0%	17484	12.0%

La table 6.1 présente les résultats de notre analyse. La colonne *WCET avec boucles bornées* montre le WCET du programme avec les bornes de boucles comme seule annotation de flot, de façon à ce que l'analyse se termine. Les autres colonnes présentent les résultats de notre analyse avec l'utilisation des automates d'annotations pour encoder les contraintes d'exclusivité décelées dans des fonctions spécifiques. La colonne *contraintes additionnelles* présente les résultats avec l'intégration des annotations sous forme de contraintes ILP supplémentaires alors que la colonne *dépliage* montre les résultats lorsque les annotations sont intégrées dans l'analyse par des automates à deux états capables de déplier le CFG.

Les résultats montrent que les deux types d'automates ont intégré correctement les annotations relatives à l'exclusivité de certaines branches et aboutissent au même gain de précision sur l'estimation de WCET. Le fait que les deux résultats soient identiques n'est pas un problème puisque cela signifie simplement qu'il n'y avait aucun gain à tirer par la méthode du dépliage de graphe. Cela nous conforte cependant dans l'idée que les deux méthodes aboutissent bien au même résultat. En réalité le seul cas qui aurait soulevé des questions aurait été que l'intégration par contraintes additionnelles

2. Peu de programmes ont été sélectionnés car nous avons fait les recherches de chemins infaisables exploitables à la main dans les codes sources des différents programmes et que nous n'avons pu détecter de tels chemins que dans ces deux programmes.

soit plus précise que la version dépliée alors qu'en théorie les chemins autorisés par le graphe déplié, dans lequel l'annotation a conservé sa localité, sont inclus dans ceux autorisés par la contrainte couplée au CFG, pour lesquels la contrainte est devenue globale.

6.3.3 Gains de précision dans les analyses bas-niveau

L'exemple de la partie 6.2 est une version simplifiée du programme *excl_mod*. Le résultat de l'analyse de WCET de ce programme est présenté dans la table 6.2. Les colonnes sont strictement les mêmes que dans la partie précédente.

Table 6.2 – Résultats de l'analyse de WCET de l'exemple de la partie 6.2 avec et sans automates.

	WCET avec	contraintes additionnelles		dépliage	
Programme	boucles bornées	WCET	Gain	WCET	Gain
excl_mod	419810	312900	25.5%	253556	39.6%

Les résultats de ces analyses montrent un premier gain entre les analyses avec automates et l'analyse initiale, et un second gain de précision du WCET entre la méthode d'intégration par dépliage et la méthode utilisant une contrainte additionnelle. Le programme *excl_mod* est composé d'une boucle simple comportant deux blocs `A` et `B` mutuellement exclusifs, suivis d'un bloc `C`, tous trois coûteux en terme de durée d'exécution.

Comme expliqué dans la partie 6.2.4, l'intégration de l'annotation d'exclusivité par la méthode de la contrainte additionnelle aboutira à l'ajout, dans le système ILP associé au CFG, de l'équation $x_{\texttt{A}} + x_{\texttt{B}} \leq x_{\texttt{backedge}}$. Cette contrainte globale garantit qu'une exécution du programme passera au maximum dans le bloc `A` ou dans le `B` autant de fois que le nombre d'itérations de la boucle. Sans cette contrainte, le WCEP du CFG passe par les deux blocs `A` et `B` à chaque itérations de la boucle. L'intégration de cette contrainte permet d'aboutir aux 25.5% de gains visibles dans la table 6.2.

L'intégration de l'exclusivité par dépliage montre un gain additionnel de 14.1% qui est principalement dû au comportement du cache d'instruction et à la pénalité d'accès à la mémoire associée à un *miss*. Il a été mentionné que les blocs `A`, `B` et `C` sont des blocs coûteux en temps d'exécution, mais plus précisément, il s'avère qu'ils remplissent chacun une des deux voies du cache d'instruction. Or, avec la politique de

remplacement LRU, dans la séquence A → B → C → A → ..., aucun accès à un de ces blocs ne correspondrait à un *hit* dans le cache et chaque bloc remplacerait alors invariablement un des deux autres blocs comme illustré sur la figure 6.9.

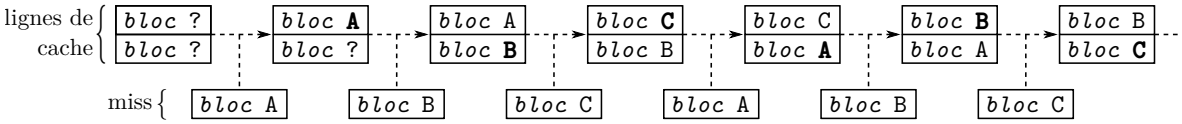


Figure 6.9 – Remplacements successifs des différentes lignes de caches par la séquence A → B → C → A → ...

Lorsque l'exclusivité est intégrée par l'ajout d'une contrainte ILP globale, le pire cas correspond à celui où l'on passe successivement par A, B, et C pendant la moitié des itérations et uniquement par C pour l'autre moitié. En terme de cache abstrait, les trois blocs seront catégorisés comme *always miss*, ce qui signifie qu'ils entraîneront une pénalité d'accès à la mémoire à chacune de leurs exécutions. Le bloc C aurait pu bénéficier d'une catégorie plus précise de type *half miss*, ce qui aurait réduit de moitié ses pénalités d'accès à la mémoire mais les méthodes utilisées (voir partie 2.3.4.2) n'offrent pas cette possibilité.

En intégrant l'exclusivité par dépliage de graphe, la séquence A → B → C → A → ... a disparu de la structure du CFG (pour s'en convaincre, il faut reprendre la figure 6.7 en considérant que le bloc 8 correspond au bloc C). Ainsi, le chemin le plus coûteux est devenu la séquence A → C → B → C → ... qui entraîne une éviction mutuelle des blocs A et B, qui resteront donc catégorisés comme *always miss*, mais qui permet au bloc C de persister dans le cache après un premier chargement. Le comportement du cache associé à cette séquence est illustré sur la figure 6.10

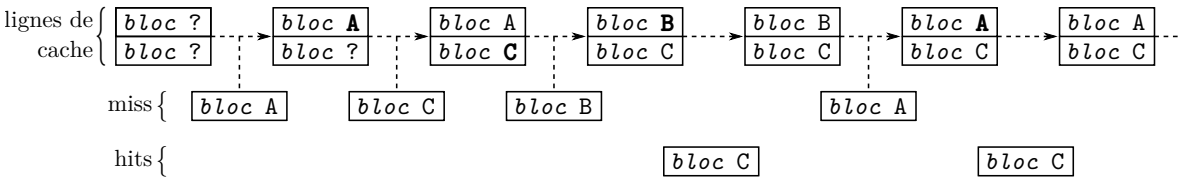


Figure 6.10 – Remplacements successifs des différentes lignes de caches par la séquence A → C → B → C → ...

Dans ce cas précis, le dépliage du CFG a donc permis de catégoriser le bloc C comme *persistent* alors que la méthode d'intégration par contrainte lui associait la catégorie *always miss*, ce qui a permis de gagner en précision lors de l'estimation du WCET.

6.3.4 Impact de la taille du programme sur la précision

Afin de tester les limites de notre approche et de considérer diverses densités et quantités de chemins infaisables, nous avons étendu l'exemple de la partie précédente à trois familles de programmes nommées `sparse_`n, `half_`n et `dense_`n où n est un entier. Le code source 6.3 montre le programme commun à ces trois familles pour un n valant 4, et la figure 6.11 présente son CFG.

```
int main() {
  for(i = 0; i < 100; i++) {
    if(...) {...} // A
    if(...) {...} // B
    if(...) {...} // C
    if(...) {...} // D
  }
}
```

Code 6.3 – Programme comportant quatre conditions dans une boucle.

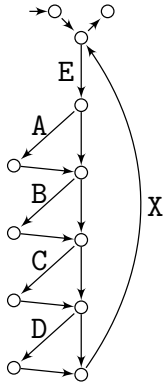


Figure 6.11 – CFG du programme 6.3.

Initialement, on trouve dans chaque itération de ce programme 2^n chemins possibles. On souhaite définir des annotations de flot permettant d'autoriser un nombre variable de chemins dans chaque itération (en interdisant par complément certains chemins infaisables). Pour cela, on a donc regroupé les programmes dans des familles, en fonction de la densité des conditions autorisées pour chaque tour de boucle, et donc du nombre de chemins faisables :

`sparse_`n : Une seule condition peut être traversée à chaque itération, soit $n+1$ chemins faisables (pour $n = 4$, les séquences autorisées sont `E` $\rightarrow$ `X`, `E` $\rightarrow$ `A` $\rightarrow$ `X`, `E` $\rightarrow$ `B` $\rightarrow$ `X`, `E` $\rightarrow$ `C` $\rightarrow$ `X` et `E` $\rightarrow$ `D` $\rightarrow$ `X`). Le nombre de conditions autorisées par tour de boucle est toujours 1 et n'est donc pas impacté par le facteur n.

`half_`n : La moitié des conditions peut être traversée à chaque itération, ce qui représente, lorsque n est pair, $\sum_{i=0}^{n/2} \binom{n}{i}$ chemins faisables, soit une valeur proche de $\frac{2^n}{2}$. Le nombre de conditions autorisées correspond dans ce cas à $n/2$, ce qui n'a vraiment de sens que pour un nombre pair de conditions.

`dense_`n : Toutes les conditions sauf une peuvent être traversées à chaque itération,

soit $2^n - 1$ chemins faisables (et par complément, un seul infaisable). Le nombre de conditions autorisées ici est $n - 1$.

Ces différentes familles de programmes ont été analysés en faisant croître le nombre de conditions n. Le graphique 6.12 montre les résultats de différentes analyses de WCET du programme `sparse_`n [3].

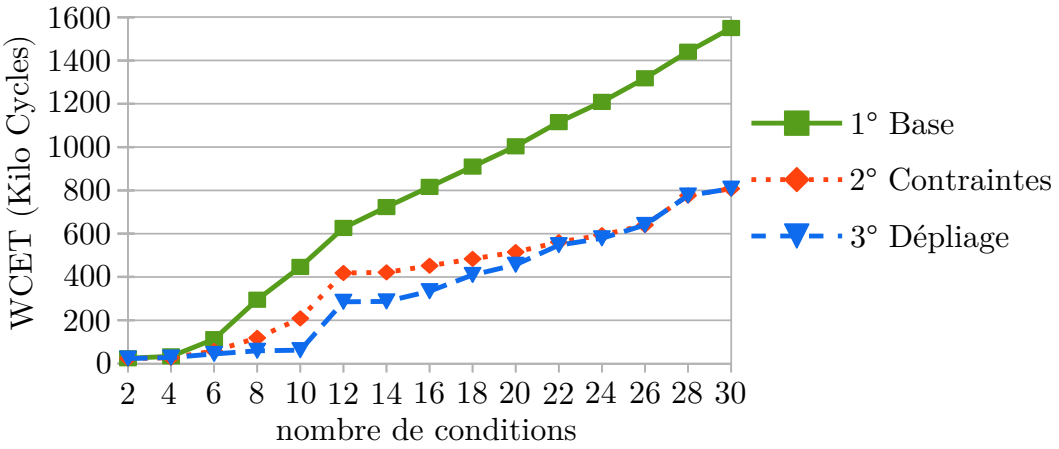


Figure 6.12 – Estimation du WCET en fonction du nombre n de conditions dans le programme `sparse_`n.

Les trois courbes présentées sur ce graphique correspondent respectivement à l'analyse de base sans l'intégration des annotations sur les conditions (courbe 1°), avec intégration des annotations par la méthode des contraintes ILP additionnelles (courbe 2°) et avec intégration des annotations par dépliage du CFG (courbe 3°).

Interprétation

La séparation de la courbe 1° avec les deux autres courbes montre le gain effectif dans la précision du WCET estimé issu de l'intégration des annotations. En effet, le WCEP initial (pour la courbe 1°) correspond naturellement à passer dans toutes les conditions dans chaque itération, ce qui n'est plus possible avec l'intégration des annotations. Par ailleurs, les courbes 2° et 3° croissent également avec le nombre de conditions n. Parmi les raison de cet accroissement, on peut citer le coût non négligeable des blocs de base correspondants aux tests des conditions, ainsi que les pénalités d'accès à la mémoire appliquées dans chaque itération à chaque bloc puisque rien n'interdit de passer dans une condition différente à chaque tour de boucle.

3. Les deux autres familles de programmes seront utilisées dans la partie 6.3.5

Entre 8 et 18 conditions, on constate un net décrochage entre les courbes 2° et 3° que l'on peut expliquer de la façon suivante. Si l'on considère que le CFG de la figure 6.11 a été déplié par une contrainte n'autorisant qu'une condition dans chaque itération, alors le chemin le plus long dans une itération de ce graphe déplié sera composé de 6 blocs, alors que le chemin le plus long lorsque le graphe n'est pas déplié est composé de 9 blocs. Avec l'augmentation du nombre n de conditions, cette différence devient plus marquée puisque le chemin le plus long, composé initialement de $2n+1$ blocs, sera réduit à $n+2$ blocs dans la version dépliée. Si on fait un parallèle avec l'exemple de la partie 6.3.3, on peut remarquer que pour une taille de cache adaptée, cette différence de taille de chemin rend possible l'apparition de cas où certains blocs du graphe déplié seront catégorisé comme *persistent* au lieu de *always miss*. Ainsi, cette modélisation plus précise des caches abstraits réduira globalement le nombre de pénalités d'accès à la mémoire ainsi que la sur-estimation du WCET.

Notons enfin que seule la famille `sparse_`n nous permettait une intégration par dépliage de graphe, et donc une comparaison des résultats des deux méthodes. L'automate permettant de déplier le CFG pour la famille `sparse_`n ne comporte que deux états alors que les automates pour les deux autres familles sont composés d'un nombre exponentiel d'états. Nous verrons dans le chapitre 7 comment il est possible d'utiliser l'ordre des conditions dans le CFG pour réduire la complexité de ces automates et avoir ainsi la possibilité d'intégrer ces annotations par dépliage de graphe.

6.3.5 Passage à l'échelle

L'intégration d'annotations l'analyse de WCET par les automates à contraintes entraîne un surcoût en terme de temps d'analyse que nous avons tenté de quantifier. Pour cela, nous avons utilisé les trois familles de programmes présentées dans la partie précédente, en augmentant jusqu'à 256 le nombre de conditions. Les résultats de ces mesures du temps d'analyse sont présentés dans la figure 6.13

Ce graphique est composé de quatre courbes montrant l'évolution de la durée d'analyse du WCET en fonction du nombre de conditions du programme analysé.

1° Cette courbe montre la durée d'analyse du WCET avec la borne de la boucle du programme comme seule annotation de flot, et sans utiliser les automates d'annotation.

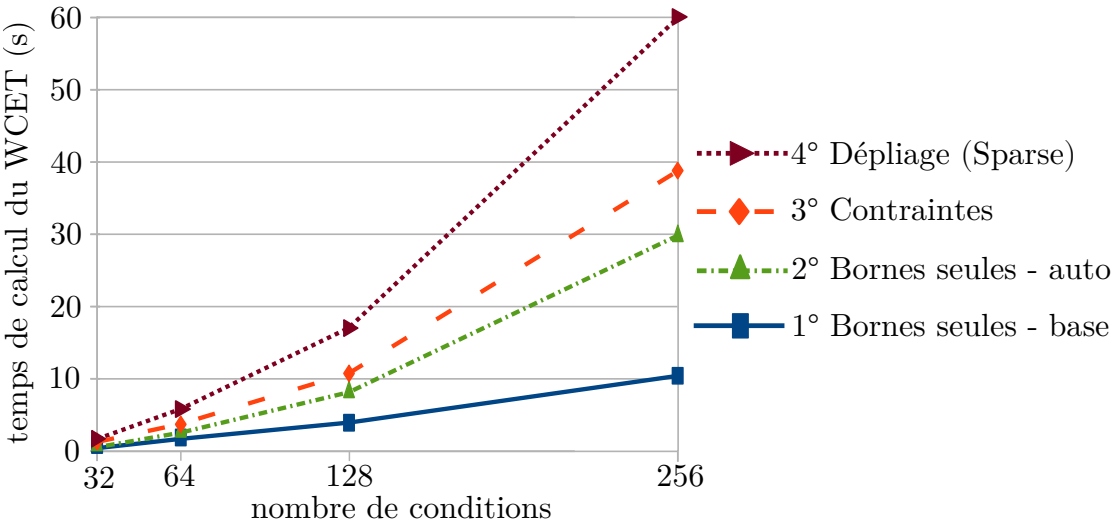


Figure 6.13 – Temps d'analyse nécessaire au calcul du WCET.

2° Cette courbe montre le temps d'analyse du WCET avec la borne de boucle du programme comme seule annotation de flot, en utilisant les automates d'annotation pour son intégration dans l'analyse.

3° Cette courbe regroupe les durées d'analyse (très similaires) des WCET pour les trois familles de programmes, avec l'intégration des diverses annotations sur les conditions sous forme de contraintes ILP additionnelles.

4° Cette courbe montre la durée d'analyse du WCET du programme `sparse_`n avec intégration des annotations sur les conditions par un dépliage du CFG.

Le coût de l'utilisation des automates pour intégrer des contraintes dans l'analyse est mis en évidence par la différence entre les courbes 1° et 2°. Cette mise en lumière nous a permis de trouver des pistes d'améliorations de l'implémentation des automates sous forme de *plug-in* dans l'outil d'analyse Otawa, afin de réduire ce surcoût de traitement.

L'allure de la courbe 3°, proche de la courbe 2° montre que l'intégration des annotations sur les conditions sous forme de contraintes additionnelles provoque un surcoût maîtrisé de la durée d'analyse globale, avec des bénéfices en terme de précision du WCET estimé qui peuvent être importants (voir partie 6.3.4).

Enfin, pour la famille de programmes `sparse_`n, le coût du dépliage de graphe pour intégrer les annotations relatives aux conditions, bien que nettement supérieur aux autres coûts, ne fait pas apparaître d'explosion exponentielle, puisque la courbe 4°

correspond en moyenne à deux fois la courbe 2°.

6.4 Le dépliage de contraintes numériques

Les différents résultats expérimentaux présentés dans la partie précédente montrent les bénéfices potentiels que l'intégration d'annotations par dépliage de graphe peut offrir, au prix d'un accroissement de la complexité de l'analyse. Ces gains de précision dans l'estimation de WCET ont motivé l'idée de construire des automates capables de déplier certaines annotations de flot qu'il est possible de rencontrer lors de l'analyse d'un programme.

Les contraintes numériques utilisées dans la méthode IPET constituent généralement des restrictions sur le nombre d'exécutions des éléments du CFG. Les équations utilisées pour représenter ces contraintes sont principalement des équations linéaires positives du premier degré :

$$(1)\ \sum_{i=1}^{n} a_i x^i \leqslant \beta, \text{ ou plus rarement } (2)\ \sum_{i=1}^{n} a_i x^i \geqslant \beta, \text{ avec } a_i, x_i, \beta \in \mathbb{N}.$$

Nous nous intéresserons principalement aux inéquations de type (1) dont la forme correspond bien à une restriction du nombre d'exécution d'éléments du CFG (x^i) alors que les inéquations de type (2) correspondent à une borne minimum d'exécutions d'éléments du CFG.

6.4.1 Principe général

Dans [9], Boudet et Comon montrent qu'il est possible de construire un automate fini capable de reconnaître les solutions d'une équation diophantienne. Cette démonstration est donc valable pour les inéquations qui nous intéressent. La construction que nous souhaitons définir s'intègre cependant dans une méthode d'analyse de WCET, et plus particulièrement dans l'expression de chemins infaisables, en interdisant des arcs spécifiques du CFG. Il conviendra donc d'adapter l'automate construit à partir de la contrainte pour qu'il permette d'interdire les mêmes chemins que la contrainte initiale.

Pour procéder à ce dépliage d'automate, nous proposons d'enrichir temporairement

les états de l'automate avec les contraintes associées au FFA initial et d'adapter la fonction de transition pour que celle ci puisse modifier les contraintes désormais contenues dans les états. Il reste ensuite à construire l'automate en suivant chaque transition et en adaptant la contrainte dans l'état cible pour chaque variable rencontrée sur l'arc emprunté. Cette modification de la contrainte permettra de prendre en compte le fait qu'on a consommé par cette transition une variable de la contrainte. Ainsi, pour chaque variable α rencontrée, il conviendra de substituer α par $\alpha + 1$ dans les contraintes de l'état cible.

6.4.2 Un exemple

Le formalisme des automates à contraintes supporte parfaitement les inéquations mentionnées par l'utilisation de contraintes et de variables. Il suffit pour cela d'attacher une variable à chaque arc concerné par l'annotation et d'associer une contrainte à l'automate.

```
<control-constraint>
 <le>
  <add>
   <mul>
    <int>3</int><count edge="A"/>
   </mul>
   <mul>
    <int>5</int><count edge="B"/>
   </mul>
   <mul>
    <int>2</int><count edge="C"/>
   </mul>
  </add>
  <int>7</int>
 </le>
</control-constraint>
```

Code 6.4 – Fichier FFX supportant une contrainte entre les arcs `A`, `B` et `C`.

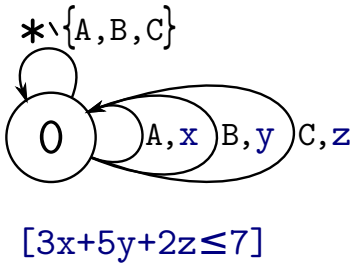


Figure 6.14 – FFA représentant l'annotation du code 6.4.

Le code 6.4 présente, dans le format FFX, une contrainte relative au nombre d'exécutions de trois arcs `A`, `B` et `C` du CFG. La contrainte ILP correspondante serait écrite $3x_A + 5x_B + 2x_C \leq 7$.

La figure 6.14 montre un automate à contraintes supportant l'annotation de flot décrite en FFX dans le code 6.4. On y retrouve les variables `x`, `y` et `z` respectivement attachées aux arcs `A`, `B` et `C` du CFG, ainsi que la contrainte exprimée à partir de ces

variables. Rappelons par ailleurs que ces variables correspondent au fait d'emprunter certains arcs du CFG, et qu'elles prennent donc leurs valeurs dans $\mathbb{N}$.

Comme expliqué dans la partie précédente, la construction de l'automate déplié correspondant à cette contrainte suppose d'enrichir les états avec les contraintes en elle-mêmes. Ce procédé est visible dans la figure 6.15 qui présente la première étape de construction de cet automate déplié.

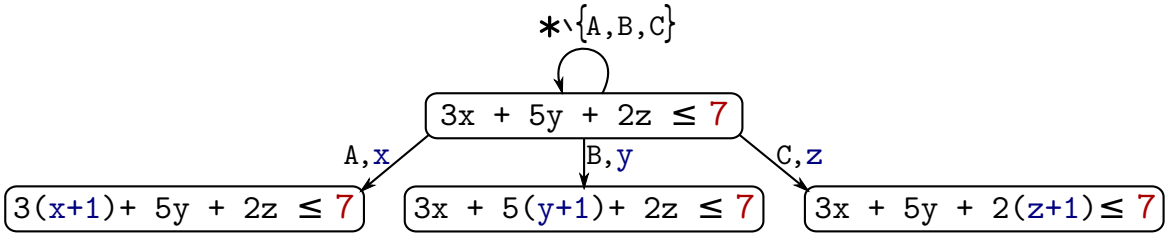


Figure 6.15 – Première étape de construction de l'automate déplié correspondant au FFA de la figure 6.14.

À partir de l'état initial, pour chaque transition supportant une variable, on construit un nouvel état dans lequel on recopie la contrainte. On modifie ensuite cette contrainte comme illustré sur la figure 6.15. Pour chaque variable α apparaissant sur la transition empruntée, on opère une substitution par $(\alpha + 1)$ dans la contrainte de l'état cible. En développant le coefficient de la variable substituée, on réduit ainsi la borne globale de la contrainte. En d'autres termes, on retire de la contrainte le coefficient correspondant à un passage par la branche associée à la variable.

Par ailleurs, cette méthode de construction nécessite des conditions d'arrêt que l'on peut définir à partir des contraintes. À l'origine, une contrainte interdit certaines combinaisons d'exécutions des arcs du CFG. Dans notre exemple, si l'arc `B` est pris une fois, alors il ne sera pas possible de le reprendre une seconde fois tout en respectant la contrainte, mais il sera toujours possible d'emprunter l'arc `C`. Avant de construire un nouvel état, il faut donc anticiper l'état de la contrainte après substitution et vérifier qu'elle reste satisfaisable en considérant que les variables qui la composent prennent leurs valeurs dans $\mathbb{N}$. Il ne sera pas question de *solveur SAT* ici, puisque pour une inéquation de type (1), il suffit vérifier si la contrainte est vraie quand toutes les variables valent zéro. Dans le cas contraire, l'état correspondant ne sera pas construit.

La figure 6.16 montre le résultat de la construction complète de l'automate déplié.

Les états sont discriminés par leurs contraintes, ce qui signifie que deux états contenant la même contrainte sont équivalents. Par exemple, le fait de prendre l'arc `A` puis

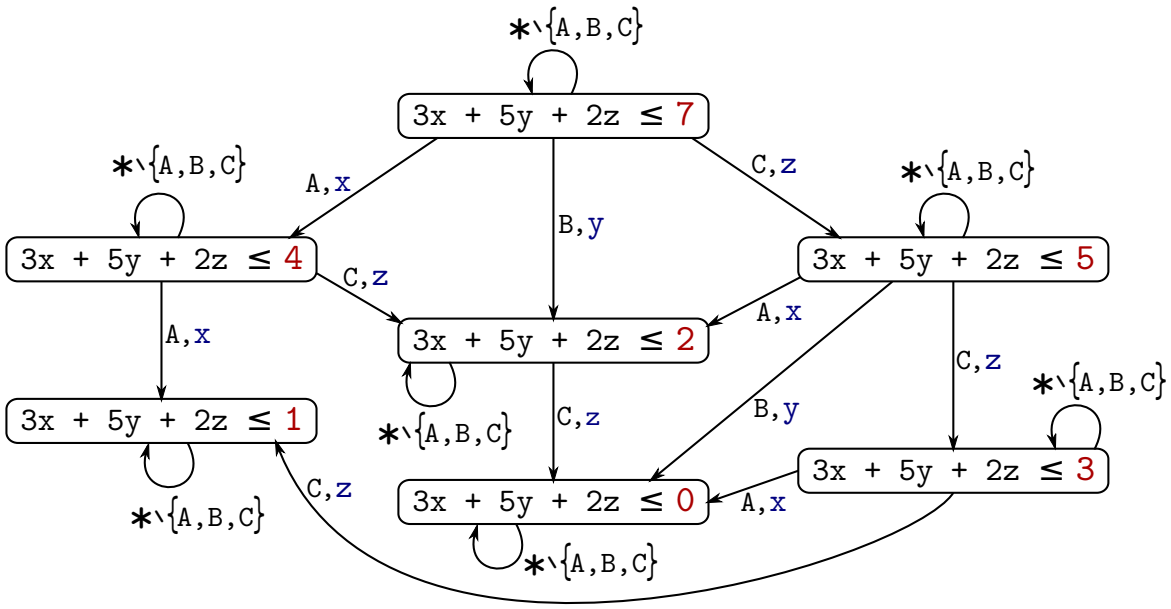


Figure 6.16 – Automate enrichi résultant du dépliage du FFA de la figure 6.14.

l'arc `C`, ou l'arc `C` puis l'arc `A` consommera le même budget sur la borne totale de la contrainte. Pour cette raison, divers états ont été fusionnés durant le processus de construction.

L'automate construit ainsi représente structurellement les séquences d'arcs autorisés par la contrainte. Afin de revenir au formalisme des FFAs, il est possible de simplifier cet automate enrichi en supprimant les contraintes et les variables qui sont devenues inutiles. La figure 6.17 montre le résultat de cette simplification.

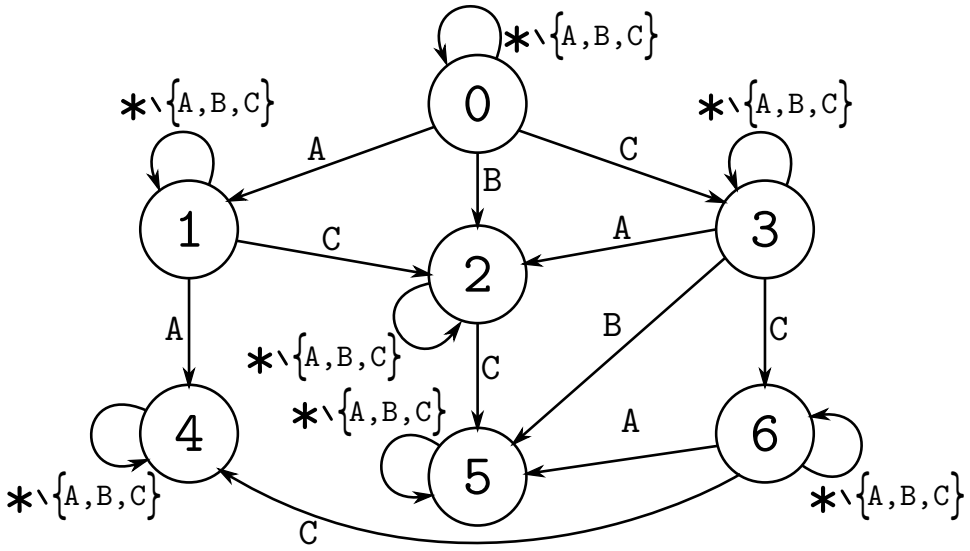


Figure 6.17 – FFA simplifié correspondant à l'automate enrichi de la figure 6.16.

6.4.3 Un programme spécifique

Nous avons expérimenté cette méthode de dépliage à partir d'une contrainte sur un programme développé spécifiquement appelé *budget*. Ce programme comporte une boucle principale dans laquelle plusieurs appels de fonctions sont effectués. Le code 6.5 représente la partie du programme correspondant à une itération de la boucle. Chaque itération est composée d'un certain nombre de conditions et d'appels à des tâches qui peuvent être effectuées en mode normal ou en mode "dégradé", c'est-à-dire plus rapidement mais moins précisément. Pour donner un exemple, si le mode normal d'une tâche consiste à trouver la valeur approchée à 10^{-9} d'un nombre, le mode dégradé de la tâche permettra d'obtenir une précision de 10^{-5}. Ainsi, avant d'autoriser l'exécution d'une tâche en mode normal, le programme vérifie qu'il sera ensuite toujours possible d'effectuer les tâches suivantes en mode dégradé.

```
assert(BUDGET >= qa + qb + qc);
if (rand()) {
  budget -= qA;
  A();
}
if (rand()) {
  if (qB + qc <= BUDGET) {
    budget -= qB;
    B();
  } else {
    budget -= qb;
    b();
  }
}
...
assert(BUDGET >= 0);
```

Code 6.5 – Extrait de code du programme *budget*

Il est possible d'exprimer cette notion de budget à respecter au moyen d'une contrainte numérique qui associe le coût des tâches en mode normal et dégradé et la valeur du budget. La variable `BUDGET` ainsi que les poids de chaque tâche que nous avons définis arbitrairement correspondent à la contrainte ILP suivante :

$$10x_A + 22x_B + 7x_b + 12x_C + 8x_c <= 30$$

À partir de ce programme et de l'annotation mettant en relation le mode des différentes tâches et le budget total alloué à chaque itération, nous avons pu comparer notre méthode d'intégration par dépliage avec l'intégration de la contrainte telle quelle dans l'analyse de WCET. En espérant trouver des gains micro-architecturaux similaires à ceux de la partie 6.3.3, nous avons testé différentes tailles de cache d'instruction à la recherche de différences entre les deux méthodes. Pour des tailles de cache adaptées à la taille de notre programme et de nos tâches (autour de 1 kilo-octet), nous avons effectivement observé des différences notables entre les deux méthodes d'intégration. La table 6.3, dont les colonnes sont strictement les mêmes que dans la tables précédentes,

présente ces résultats.

Table 6.3 – Résultats de l'analyse de WCET du programme budget.

Programme	WCET avec boucles bornées	contraintes additionnelles		dépliage	
		WCET	Gain	WCET	Gain
budget	9415	7390	21.5%	7090	24.7%

6.5 Conclusion

Ce chapitre nous a permis de comparer les deux méthodes d'intégration de contraintes numériques dans une analyse de WCET au moyen des automates présentés dans cette thèse.

Nous avons montré sur un premier exemple comment il était possible d'intégrer des contraintes numériques dans une analyse de WCET par un dépliage partiel du CFG, en détaillant les différentes étapes du processus et l'utilisation des opérations présentées dans le chapitre 4. Cette méthode a été comparée avec la méthode traditionnelle d'intégration reposant sur l'ajout de nouvelles contraintes ILP, et nous avons mis en évidence des gains de précision dans l'analyse lors de l'utilisation de la méthode de dépliage. Ce phénomène a été expliqué par l'impact du dépliage de graphe sur les analyses bas niveau relatives à la micro-architecture.

Les limites et le passage à l'échelle de cette approche de dépliage ont été testées, et des problèmes de complexité ont été mis en évidence, notamment dans le dépliage de certaines familles de programmes en raison de la nature des contraintes. Nous aborderons cependant ce problème dans le prochain chapitre où nous utiliserons l'ordre des arcs dérivé du CFG pour simplifier les annotations initiales et parvenir à les intégrer dans une analyse de WCET par dépliage de graphe.

Pour finir, nous avons présenté une méthode systématique de dépliage de contraintes ainsi qu'un programme expérimental destiné à faire la preuve de concept de cette approche. Les résultats expérimentaux nous ont permis de confirmer la tendance de la méthode de dépliage à être plus précise que la méthode traditionnelle.

"Sire, on en a gros !"

— Perceval le Gallois

7

Extension à une nouvelle classe d'annotation

Sommaire

7.1 Introduction

Les HFAs présentés dans cette thèse permettent d'intégrer des annotations de flot dans une analyse de WCET, avec pour objectif d'en améliorer la précision. Pour cela, les annotations fournies, sont traduites en chemins infaisables et représentées sous forme d'automates qui les interdisent spécifiquement [1]. Ces automates sont ensuite fusionnés

1. en acceptant tous les autres chemins par complément

avec le CFG et le système ILP associé, ce qui rend inaccessibles ces chemins et participe à la réduction du pessimisme de l'estimation du WCET. Ces automates sont donc bien adaptés à la prise en compte de l'infaisabilité de certains de chemins dans l'analyse, mais dans la pratique, les seuls chemins infaisables rencontrés sont ceux relatifs aux bornes de boucles, souvent fournis par des outils (comme oRange), ou ceux associés à des blocs spécifiques (souvent des conditions), qui dépendent de la sémantique du code et qui sont fournis dans ce cas par un expert.

L'outil PathFinder a pour objectif de détecter des chemins infaisables dans un programme binaire. Plus précisément, il est capable de détecter des conflits entre plusieurs arcs du CFG. Ainsi, lorsqu'une liste d'arcs en conflits est détectée, cela signifie que l'ensemble des chemins du CFG qui passent par tous les arcs de cette liste sont des chemins infaisables. Nous avons donc décidé de définir une classe spécifique pour ces conflits afin de les intégrer dans une analyse de WCET. Pour cela, nous avons enrichi en premier lieu le langage d'annotation FFX, qui s'y prête bien puisqu'il hérite naturellement de la propriété extensible du format XML. Ainsi, il nous a été possible d'ajouter une nouvelle annotation en ayant pris soin de respecter les règles et la structure du format actuel, notamment en ce qui concerne la hiérarchie des contextes commune aux différentes annotations déjà supportées. Puis, à partir de cette nouvelle classe d'annotations, nous avons construit des HFAs afin d'intégrer ces conflits dans une analyse de WCET. La possibilité d'obtenir un CFG partiellement déplié est apparue de la fabrication de certains HFAs et nous nous sommes donc intéressés à nouveau aux différences de précision du WCET estimé qui pouvait découler de ce dépliage. Durant ce processus, nous avons fait face à des problèmes de complexité et nous avons donc étudié les propriétés de cette classe d'annotation pour les résoudre.

Cette nouvelle classe d'annotation, son expression en FFX et en automates ainsi que les résultats expérimentaux associés ont été présentés dans l'article [45].

7.2 Les conflits

Cette section présentera la notion de conflit en elle-même, issue des résultats de l'outil d'analyse PathFinder. Pour commencer, nous présenterons donc succintement cet outil et son fonctionnement, puis nous définirons précisément les conflits et leur traduction en une annotation FFX.

7.2.1 L'outil PathFinder

PathFinder est un outil développé à l'IRIT présenté dans l'article [56]. Il a pour objectif de trouver des chemins infaisables dans les programmes binaires en cherchant des conflits entre les différents arcs du CFG. Pour cela, l'analyse effectuée, basée sur l'interprétation abstraite, parcourt le CFG depuis l'entrée du programme et construit des conjonctions de prédicats sur les registres et les données en mémoire. La satisfaisabilité de ces conjonctions est vérifiée au fur et à mesure du parcours grâce à un solveur SMT [2] pour détecter dans le CFG des ensembles de chemins infaisables. L'analyse s'efforce également de réduire au maximum le nombre d'arcs réellement responsables des conflits détectés, afin de rendre ce résultat plus facilement exploitable par d'autres analyses. Ainsi, un conflit détecté correspondra à un ensemble d'arcs du CFG qui ne peuvent pas être empruntés lors d'une même exécution du programme. Le résultat d'une analyse d'un binaire par PathFinder est donc une liste d'ensembles d'arcs, c'est-à-dire une liste de conflits.

7.2.2 La notion de conflit

Les conflits découverts par PathFinder (ou un outil similaire) s'expriment sous forme de listes d'arcs du CFG qui ne peuvent pas être tous empruntés dans une même exécution.

```
<conflict>
  <!-- identifiant d'arc ou de bloc 1 -->
  <!-- ... -->
  <!-- identifiant d'arc ou de bloc N -->
</conflict>
```

Code 7.1 – Forme générale d'un conflit

```
<conflict>
  <edge src="0x8368" dst="845C">
  <edge src="0x8480" dst="8524">
  <edge src="0x868C" dst="87D0">
</conflict>
```

Code 7.2 – Exemple d'annotation

Le code 7.1 montre la forme générale de la balise associée à un conflit. Imbriqués dans cette balise, on trouve les différents éléments qui sont en conflit. On y trouvera habituellement des arcs ou des blocs du CFG représentés par leurs adresses physiques (c'est à dire les adresses physiques des blocs auxquels ils sont rattachés pour les arcs) mais le format FFX offre également la liberté d'utiliser des références vers des *identifiants* [3] afin de définir les blocs ou arcs concernés ailleurs dans le fichier FFX.

2. Satisfiability Modulo Theory
3. Voir `IDENTIFICATION` dans l'appendice A

Le code 7.2 illustre un conflit entre trois arcs du CFG, représentés par les adresses physiques de leurs blocs source et destination. Cette annotation indique que toute exécution qui emprunterait au moins une fois chacun des trois arcs mentionnés est infaisable. Un intérêt direct pour le domaine de l'analyse de WCET est que si une exécution passant par ces trois arcs s'avérait être le WCEP trouvé par l'analyse courante, l'intégration de cette annotation de conflit ne pourrait qu'améliorer la précision de l'analyse de WCET.

7.2.2.1 Une annotation indépendante du nombre d'exécution

Il est possible d'exprimer la plupart des conflits découverts à partir de contraintes ILP associées aux arcs (ou aux blocs) concernés. Cependant, la notion de conflit n'est fondée que sur l'absence dans les exécutions d'au moins un des ces arcs, sans prendre en compte le nombre de fois ou les différents arcs sont rencontrés.

```
if(X) // IF_A
  tst=0; // A
while(Y)
  if(tst==1) // IF_B
    foo(); // B
  ...; // instructions sans effets sur tst
```

Code 7.3 – Conflit impliquant un bloc dans une boucle

Une recherche de chemins infaisables sur le code 7.3 pourrait mettre en évidence un conflit entre les blocs `A` et `B` (ou de façon équivalente entre les arcs `IF_A`→`A` et `IF_B`→`B`). Le bloc `B` est dans une boucle mais le nombre d'exécutions de cette boucle n'affecte pas le conflit découvert : les exécutions dites infaisables sont toutes celles qui passent au moins une fois par `A` et au moins une fois par `B`.

Le code 7.4 exprime le conflit entre les blocs `A` et `B` au format FFX. Le nombre d'exécutions de `B` n'est pas mentionné mais il est équivalent en réalité à la borne de la boucle. Si on voulait exprimer le conflit numériquement, il serait nécessaire de connaître et d'utiliser cette borne (n) dans une contrainte comme :

$$n \times \mathtt{A} + \mathtt{B} \leq n$$

```
<conflict>
  <block "A"/>
  <block "B"/>
</conflict>
```

Code 7.4 – Conflit entre deux blocs

7.2.3 Le contexte d'un conflit

L'imbrication des balises en FFX représente la hiérarchie des contextes d'appels, des boucles et des itérations. Lorsqu'un conflit est détecté, il convient donc d'imbriquer la balise <`conflict`> correspondante dans son contexte de validité.

L'outil PathFinder et les automates d'annotations ne supportent tous deux qu'une version *inlinée*[4] des CFGs des programmes analysés. De plus, PathFinder effectue son analyse sur des programmes binaires, et décèle des conflits directement entre les arcs du CFG représentés par leurs adresses physiques. Or, comme expliqué dans la partie 3.4.1, ces adresses ne sont pas uniques dans un CFG *inliné*. Il convient donc de distinguer les contextes d'appels des éléments en conflit pour fournir une annotation cohérente. D'autre part, on pourrait également utiliser des informations de localisation sous forme de lignes dans des fichier de code source qui seraient ensuite traduites en adresses physiques grâce aux informations de débogage, mais cette traduction est la même que celle décrite dans la partie 5.4, et, si ce mode d'adressage convient bien aux annotations fournies par un humain, PathFinder fournit directement les adresses physiques des arcs du CFG. Dans la suite, on se concentrera donc sur l'étude des conflits en considérant que les éléments d'un conflit sont des arcs représentés par leurs adresses physiques, mais les résultats obtenus seraient tout aussi valables si on avait comme éléments en conflit des blocs adressés par leurs numéros de ligne dans un fichier source.

Le contexte d'un conflit n'est pas aussi rigide que les autres contextes existants (boucles, fonctions, ...), dans le sens ou il ne dépend pas directement de la structure du programme, mais plutôt de la portion du programme dans laquelle il est valide, qui correspondra au contexte le plus précis englobant tous les arcs du conflit. Ce contexte commun à tous les arcs est nécessaire, mais pas suffisant, car chaque arc du conflit possède également un contexte propre, qui sera parfois (souvent) plus précis que le contexte englobant trouvé. On voit ici que le contexte d'un conflit est particulier puisqu'il traverse les autres types de contextes tout en autorisant les éléments à posséder leurs contextes propres, c'est pourquoi on illustrera son fonctionnement au travers de divers exemples.

4. Graphe dans lequel on intègre les sous-graphes correspondants aux différentes fonctions appelées.

7.2.3.1 Exemples de conflits

On considérera dans la suite divers conflits apparaissant dans les contextes structurels qu'il est possible de définir à partir du code 7.5.

```
void foo(){
  ...
}
int main(){
  ...
  // appel C1
  foo();
  ...
}
```

Code 7.5 – Exemple de code source

```
<function name="main">
  <conflict>
    <edge "A">
    <edge "B">
  </conflict>
</function>
```

Code 7.6 – Conflit dans le `main`

```
<function name="main">
  <call name="C1">
    <function name="foo">
      <conflict>
        <edge "A">
        <edge "B">
      </conflict>
    </function>
  </call>
</function>
```

Code 7.7 – Conflit dans la fonction `foo`

Considérons le cas le plus simple : un conflit entre deux arcs dans le programme principal. Le contexte de validité de ce conflit est le programme principal en lui-même. Le code 7.6 illustre l'annotation en FFX dans laquelle le conflit est imbriqué dans le contexte de la fonction `main` englobant les deux arcs. De même, si les deux arcs en conflit apparaissent dans la fonction `foo` appelée via `C1` par le programme principal, la portion la plus précise du programme dans laquelle le conflit sera valide correspondra au contexte de la fonction `foo` en elle-même. Le code 7.7 montre l'annotation FFX correspondante et l'imbrication du conflit dans le contexte de la fonction `foo`, lui même imbriqué dans son appel `C1` et dans la fonction `main`.

Considérons maintenant que le premier arc du conflit apparaisse dans le programme principal et que le second arc soit dans la fonction `foo` appelée via `C1`. La portion du programme dans laquelle le conflit est valide correspond au `main`, puisque le premier arc est dans le programme principal, et qu'il est nécessaire d'avoir un contexte englobant les deux arcs en même temps. La solution fournie dans le code 7.6 semble convenir, mais il manque en réalité le contexte propre à l'arc `B`, car en l'état, le conflit interdit toute exécution qui contient au moins un arc `A` et un `B`. Or le conflit porte uniquement sur l'arc `B` s'il est rencontré lors de l'appel `C1`. Il est donc nécessaire de préciser le contexte spécifique à l'arc `B`, comme illustré dans le code 7.8.

Le contexte spécifique à un arc du conflit sera appelé *contexte interne* au conflit dans la suite, pour le différencier du contexte de validité (ou contexte englobant) du conflit en lui même.

```
<!-- contexte de validité -->
<function name="main">
  <conflict>
    <!-- premier arc -->
    <edge "A">
    <!-- second arc dans son
    contexte interne -->
    <call name="C1">
      <function name="foo">
        <edge "B">
      </function>
    </call>
  </conflict>
</function>
```

Code 7.8 – Conflit entre un arc du `main` et un arc de la fonction `foo`, avec précision du contexte interne du second arc.

```
<conflict>
  <!-- premier arc dans son
  contexte interne -->
  <function name="main">
    <call name="C1">
      <function name="foo">
        <edge "A">
      </function>
    </call>
  </function>
  <!-- second arc dans son
  contexte interne -->
  <function name="main">
    <call name="C1">
      <function name="foo">
        <call name="C2">
          <function name="bar">
            <edge "B">
          </function>
        </call>
      </function>
    </call>
  </function>
</conflict>
```

Code 7.9 – Conflit entre deux arcs avec leurs contextes internes complets.

7.2.3.2 Recherche du contexte englobant

Il est possible d'utiliser une méthode systématique pour trouver le contexte englobant le plus précis d'un conflit particulier, en conservant les contextes internes de chaque arc lorsque c'est nécessaire. Le processus peut être décrit ainsi :

i) Dans un premier temps, il faut définir la balise <`conflit`> à la racine, et pour chaque élément du conflit, fournir de façon exhaustive son contexte d'appel, c'est-à-dire l'ensemble des appels de fonctions, des boucles et des itérations permettant de discriminer l'élément concerné de ses répliques potentielles dans le CFG (répliques qui sont dues à la propriété *inlinée* de ce graphe).

ii) On peut ensuite fusionner les balises des contexte communs, de façon à ne jamais avoir au même niveau deux contextes équivalents.

iii) Enfin, il est possible d'extraire du conflit tous les contextes qui sont communs à l'ensemble des arcs qui le composent, ou en d'autres termes, on peut intervertir

la balise <conflict> avec son enfant [5] si celui-ci est unique et que ce n'est pas un des élément du conflit (c'est-à-dire si c'est un contexte).

```
<conflict>
  <!-- contextes internes fusionnés -->
  <function name="main">
    <call name="C1">
      <function name="foo">
        <!-- premier arc -->
        <edge "A">
        <!-- second arc dans son
        contexte interne -->
        <call name="C2">
          <function name="bar">
            <edge "B">
          </function>
        </call>
      </function>
    </call>
  </function>
</conflict>
```

Code 7.10 – Conflit du code 7.9 après fusion des contextes internes communs aux deux arcs.

```
<!-- contexte de validité du conflit -->
<function name="main">
  <call name="C1">
    <function name="foo">
      <!-- balise de conflit injectée -->
      <conflict>
        <!-- premier arc -->
        <edge "A">
        <!-- second arc dans son
        contexte interne -->
        <call name="C2">
          <function name="bar">
            <edge "B">
          </function>
        </call>
      </conflict>
    </function>
  </call>
</function>
```

Code 7.11 – Conflit du code 7.10 après extraction des contextes internes communs aux deux arcs.

Le point i) est illustré par le code 7.9. La balise du conflit a été placée à la racine et on a défini pour chaque arc du conflit son contexte d'appel exhaustif.

Afin d'appliquer le point ii) de la méthode systématique, il faut trouver les contextes internes communs aux deux arcs du conflit. La fonction `main`, ainsi que l'appel `C1` à la fonction `foo` apparaissent aux mêmes niveaux dans le code 7.9. Il est donc possible de fusionner leurs contenus en unifiant ces trois balises. Le résultat de cette étape est présenté dans le code 7.10.

Reste ensuite à transformer le contexte interne commun à tous les arcs en un contexte de validité pour le conflit, comme expliqué dans le point iii). Le code 7.10 montre le résultat de cette étape, où la balise <conflict> a bien été injectée dans son contexte de validité le plus précis.

Notons que les balises externes au conflit pourront désormais être fusionnées avec les autres annotations du fichier FFX et que le conflit a été grandement simplifié dans le processus puisque seul l'arc `"B"` possède toujours un contexte interne.

5. En XML, on parle d'enfant (resp. de parent) pour désigner une balise qui est contenue dans (resp. qui en contient) une autre.

Remarque : Le point i) requiert de fournir le contexte d'appel complet de chaque arc, mais en réalité, les contextes qui permettent de discriminer deux éléments d'un CFG *inliné* sont les appels de fonctions (les <`call`>) et les itérations spécifiques des boucles. Les balises des fonctions, tout comme les balises de boucle sont liées à des éléments déjà répliqués dans le CFG le cas échéant. Toutefois, la balise <`call`> du format FFX ne peut contenir qu'une balise <`fonction`>, et la balise <`iteration`> ne peut apparaître que dans la balise de la boucle à laquelle elle est associée. Pour ces raisons, ces quatre types de balises sont nécessaires pour spécifier le contexte d'appel complet d'un élément d'un CFG *inliné*.

7.2.3.3 Le cas des boucles

Lorsque PathFinder est utilisé sur un programme comportant des boucles, l'analyse est capable de fournir des conflits relatifs à toute itération de la boucle, mais pas à une itération précise. En effet, l'utilisation de l'interprétation abstraite consiste pour les boucles à construire des points fixes des états abstraits, ce qui limite les conflits qu'il est possible de détecter aux types suivants :

1. Un conflit au sein d'une même itération qui sera valide pour chaque itération de la boucle.
2. Un conflit entre un arc d'une itération quelconque de la boucle et un arc extérieur.
3. Un conflit entre un arc précédant la boucle et un arc de la première itération ou entre un arc de la dernière itération et un arc suivant la boucle.

En d'autres termes, il n'est pas possible de détecter un conflit impliquant une itération précise de la boucle (et un arc extérieur à la boucle par exemple).

Le point 1. sera traduit par l'imbrication du conflit dans une balise <`iteration`> avec `"*"` comme numéro d'itération, pour signifier que le conflit vaut pour chaque itération de la boucle.

Dans le point 2., l'arc interne à la boucle qui nous intéresse ne doit pas appartenir à une itération spécifique. En d'autres termes, il n'est pas nécessaire ici de préciser l'itération à laquelle cet arc doit appartenir, mais simplement que son contexte interne est la boucle en elle-même. De plus, comme indiqué précédemment en remarque, les contextes de boucle seuls ne sont pas requis pour discriminer les éléments dupliqués d'un CFG *inliné*, ce qui veut dire qu'il n'est pas nécessaire de spécifier un contexte

interne de boucle si celui ci n'est pas accompagné d'une itération précise. Il en résulte que le code 7.6 correspondrait parfaitement au conflit décrit dans ce point 2.

Enfin, l'item 3. nécessite l'utilisation de contextes internes spécifiques permettant d'isoler l'itération précise à laquelle appartient l'arc de la boucle en conflit.

7.2.4 Un premier exemple

Comme précisé dans la partie 7.2.2.1, la notion de conflit est indépendante du décompte des arcs qui la composent. Elle interdit simplement les chemins du graphe qui empruntent tous les arcs spécifiés dans le conflit. Il semble donc judicieux d'utiliser la forme dépliée des automates pour exprimer ce type d'annotation.

Considérons un programme composé de deux conditions successives. Le CFA de ce programme est présenté sur la figure 7.1.

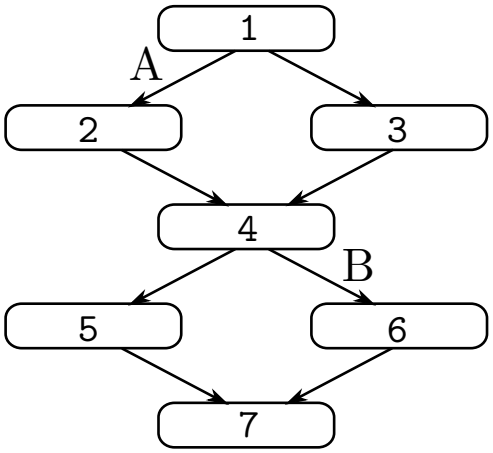


Figure 7.1 – CFA d'un programme composé de deux conditions successives.

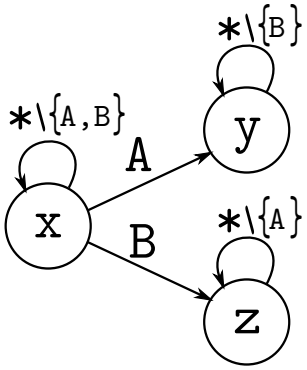


Figure 7.2 – FFA d'un conflit entre deux arcs.

Un conflit trivial entre les arcs `A` et `B` de ce CFA correspond en FFX au code 7.6 de la page 118.

La construction d'un automate équivalent à ce conflit consiste à capturer chaque arc concerné jusqu'à un dernier état correspondant à un état *poubelle*, soit un état qui ne serait pas final. Dans les FFAs, cela revient à ne pas construire ce dernier état et à interdire les transitions qui devaient y mener. La figure 7.2 montre le FFA correspondant au conflit mentionné ici. Cet automate accepte donc l'alphabet complet depuis son état initial (x) et transite, lorsque l'arc `A` (resp. `B`) est rencontré, vers l'état (y) (resp. (z)) dans lequel l'autre arc du conflit est interdit. On notera que dans les

états (y) et (z), il n'est pas interdit d'emprunter à nouveau l'arc par lequel on transité vers cet état.

En faisant un produit entre ce FFA et le CFA de la figure 7.1, on aura donc intégré l'annotation comme une restriction structurelle du CFA. Le résultat de ce produit est présenté sur la figure 7.3.

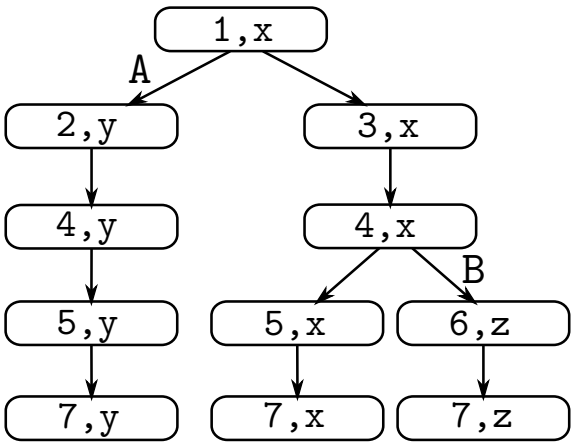


Figure 7.3 – CFA résultant du produit du CFA de la figure 7.1 et du FFA de la figure 7.2.

Pendant cette opération de produit, le passage par l'arc `A` fait transiter le FFA dans l'état (y), ce qui interdit par la suite d'emprunter l'arc `B`. Inversement, le passage par l'arc `B` fait transiter le FFA dans l'état (z), dans lequel l'arc `A` est interdit.

Un premier problème apparaît dans cette approche puisque s'il est naturel que le procédé duplique certains blocs (ici les blocs `4`,`5` et `7`) en raison des multiples états qui composent le FFA, certaines duplication semblent superflues. L'état `7,z` par exemple est séparé de l'état `7,x` pour mémoriser le fait que l'arc `B` a été emprunté pour arriver sur cet état et que l'arc `A` est donc désormais interdit. Cette information est cependant obsolète à cause de l'ordre des arcs dans le CFG.

D'autre part, le conflit mentionné ici ne concerne que deux arcs, mais rien ne limite à priori le nombre d'arcs en conflit – l'outil PathFinder fournit parfois des conflits relatifs à une vingtaine d'arcs du CFG. Or le FFA correspondant au conflit est déjà composé de trois états [6]. Pour avoir une idée de la complexité de cette représentation, on présente un FFA supportant une annotation de conflit impliquant trois arcs sur la figure 7.4.

Dans le cas d'un conflit entre deux arcs, le FFA dupliquait le dernier bloc (le

6. Ce nombre correspond en réalité à 2^2 états moins l'état poubelle qu'on ne représente pas.

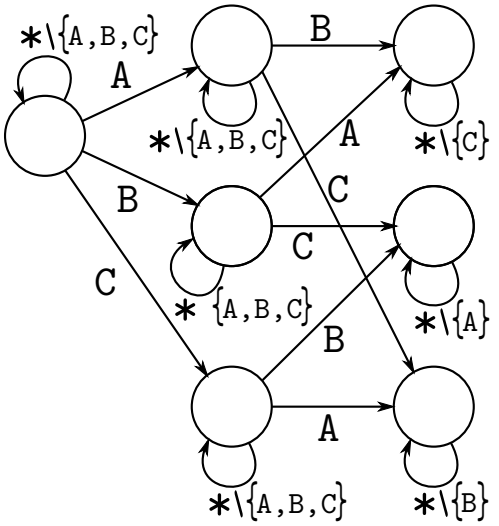


Figure 7.4 – FFA d'un conflit entre trois arcs.

bloc 7) pour chacun de ses états (trois fois). De plus, la taille de notre CFA n'était pas représentative d'un vrai programme et dans un CFG de grande taille, tous les blocs situés après ces deux conditions auraient également été dupliqués autant de fois. En utilisant le même procédé pour un conflit entre trois arcs, on risquerait de répliquer non plus trois fois, mais sept fois tous ces blocs, ce qui pourrait poser des problèmes de performances aux analyses de WCET qui suivraient. Dans tous les cas, la limite du nombre d'arcs en conflit qu'il sera possible d'intégrer à l'analyse par dépliage sera rapidement atteinte.

L'inconvénient majeur de cette approche est donc l'augmentation exponentielle du nombre d'états du FFA (qui correspond à $2^n - 1$, avec n le nombre d'arcs en conflit), accompagnée d'autant de duplications potentielles de nombreux blocs du CFG. Par ailleurs, nous nous étions déjà heurtés à ce problème dans le chapitre 6 où nous présentions trois familles de programme parmi lesquelles on trouvait la famille `dense_`n. Ces programmes composés d'un nombre variable de conditions successives étaient accompagnés d'une annotation indiquant que le chemin qui empruntait toutes les conditions était infaisable, ce qui correspond bien à la notion de conflit définie dans ce chapitre.

Pour venir à bout de ce problème, nous proposons d'utiliser l'ordre des arcs dans le CFG, afin d'exprimer une contrainte plus faible mais potentiellement moins explosive en terme de nombre d'états.

7.2.5 La notion de conflit ordonné

Dans l'exemple de la partie précédente, le FFA de la figure 7.2 interdisait de prendre l'arc `B` du conflit après l'arc `A`, et inversement. Or la structure du programme rendait impossible un chemin empruntant l'arc `B` puis l'arc `A`. En d'autres termes, il n'existe pas dans le CFG d'occurrence de l'arc `A` après l'arc `B`. Seule une moitié du conflit est donc pertinente, celle qui précise qu'il est interdit de prendre l'arc `B` après avoir emprunté l'arc `A`, puisque l'autre moitié du conflit correspond précisément à l'ordre des arcs dans le CFG. Ainsi, si on réussit à garantir que les arcs mentionnés dans le conflit ne peuvent apparaître que dans un seul ordre, il est possible de n'exprimer qu'une partie du conflit, celle qui peut réellement se produire, sans perdre de précision [7].

La figure 7.5 montre comment il est possible de reconstruire le conflit complet à partir du conflit partiel (la partie du conflit correspondant à un séquence d'arcs possible dans le CFG) et de la propriété exprimant précisément l'ordre des arcs dans le CFG.

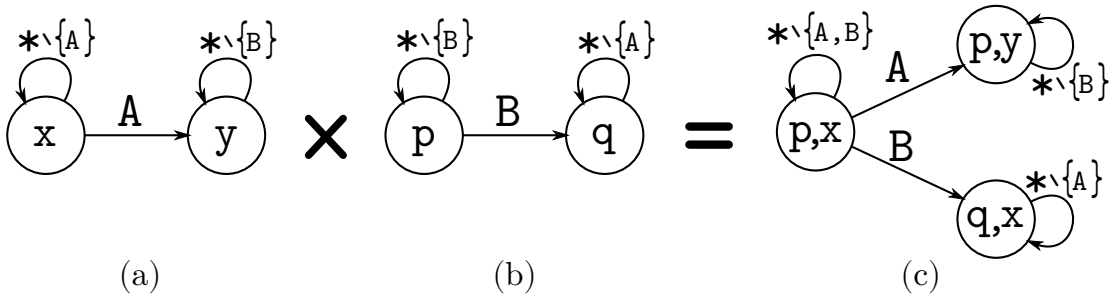


Figure 7.5 – Produit entre un conflit partiel (a) et un FFA (b) encodant la contrainte d'ordre des arcs issue de la structure du CFG. Le résultat (c) correspond au conflit complet de la figure 7.2.

Il s'avère que l'outil de détection de chemin infaisable PathFinder, capable de détecter des conflits entre certains arcs du CFG, nous garantit également que la liste d'arcs fournie ne peut apparaître que dans cet ordre dans le CFG. Il nous est donc possible d'exprimer une contrainte plus faible que le conflit complet, à savoir le conflit partiel dans lequel les arcs apparaissent dans l'ordre indiqué, et ce sans perte de précision.

Si on fait un produit entre le conflit partiel sous forme de FFA de la figure 7.5a et le CFA de la figure 7.1, on obtient le résultat illustré sur la figure 7.6.

Dans ce CFA, l'état `7,z` a disparu, ce qui ne laissera que deux branches distinctes au lieu de trois dans le CFG. Par ailleurs, le FFA du conflit partiel en lui-même a été

7. On parle ici de perte de précision quand on affaiblit une contrainte et que des chemins infaisables qui étaient interdits par la contrainte initiale redeviennent autorisés.

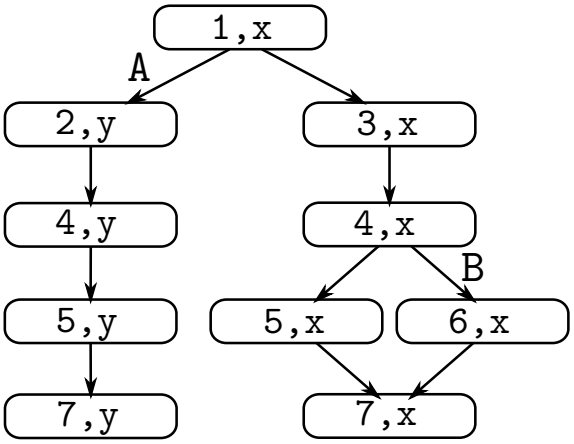


Figure 7.6 – CFA résultant du produit du CFA de la figure 7.1 et du FFA de la figure 7.5a.

considérablement réduit puisque, pour n arcs en conflit, il est seulement composé de $n-1$ états (contre 2^n-1 états pour le conflit complet).

Ainsi, l'utilisation de l'ordre des arcs permet de réduire la complexité de l'intégration des conflits par dépliage de graphe.

L'annotation de conflit ordonné en FFX

La notion d'ordre est absente du format FFX puisque dans un fichier de ce type, les annotations qui apparaissent au même niveau de la hiérarchie XML sont indépendantes. Nous avons donc ajouté un attribut `"ordered"` à la balise <`conflict`> qui permet de préciser si un conflit porte sur les arcs qui le composent dans l'ordre précis où ils sont fournis, auquel cas ils peuvent être traités en utilisant le conflit partiel présenté dans cette partie. Cela ne signifie pas que cet ordre des arcs est le seul ordre possible dans le CFG, mais seulement qu'il est possible de n'intégrer que le conflit partiel dans l'analyse. S'il s'avère que cet ordre d'exécution n'est pas le seul possible dans le CFG, alors l'annotation utilisée sera simplement moins précise que le conflit complet, mais en contrepartie, l'intégration se fera par dépliage de graphe, avec les perspectives de gains qui ont déjà été mis en évidence dans le chapitre 6.

Le code 7.12 montre un exemple de conflit qui peut être traité comme un conflit partiel en suivant l'ordre des arcs fournis. L'attribut `"ordered"` peut prendre comme valeurs `"yes"` ou `"no"`, la dernière étant la valeur par défaut.

```
<conflict ordered="yes">
  <block "A"/>
  <block "B"/>
</conflict>
```

Code 7.12 – Conflit entre deux blocs

7.3 Expériences

Dans l'article [45], nous comparons les WCET estimés obtenus par deux méthodes différentes sur la suite de programme de Mälardalen [23] et sur deux programmes générés à partir du langage Esterel [7]. La première méthode correspond à l'intégration des conflits par dépliage de graphe présentée dans la partie précédente, et la seconde, fondée sur l'approche présentée dans l'article [55], est une méthode de génération de contraintes ILP à partir des balises <`conflict`>. La table 7.1 présente un extrait de ces résultats.

Table 7.1 – Résultats de l'analyse de WCET avec intégration de conflits

	Nombre de	gain de WCET – simple arch.		gain de WCET – ARM9 et cache	
Programme	conflits	Contraintes	Dépliage	Contraintes	Dépliage
Mälardalen : Programmes courts					
cnt	5	0.00 %	0.00 %	0.00 %	0.00 %
cover	3	6.95 %	6.95 %	**0.01 %**	**0.25 %**
crc	8	0.50 %	0.50 %	**4.10 %**	**9.70 %**
expint	5	0.00 %	0.00 %	**0.00 %**	**0.09 %**
fibcall	1	0.72 %	0.72 %	0.32 %	0.32 %
fir	1	0.00 %	0.00 %	**3.37 %**	**7.45 %**
select	11	0.16 %	0.16 %	0.09 %	0.09 %
sqrt	10	0.40 %	0.40 %	0.04 %	0.04 %
Mälardalen : Programmes larges					
statemate	71	2.77 %	*ED*	1.00 %	*ED*
ud	1	1.17 %	1.17 %	1.08 %	1.08 %
nsichneu	7684	0.00 %	*ED*	0.00 %	*ED*
ludcmp	3	0.00 %	0.00 %	0.00 %	0.00 %
Programmes d'Esterel					
runner	185	9.84 %	*ED*	9.12 %	*ED*
abcd	274	3.01 %	*ED*	5.17 %	*ED*

ED : Échec de Dépliage

Les programmes mentionnés ont été compilés sans optimisation (-O0) pour l'architecture *armv5t*. L'outil PathFinder a été utilisé sur les programmes et a été capable de détecter un certain nombre de conflits qu'on retrouve dans la seconde colonne. Nous avons ensuite analysé ces programmes avec une première architecture triviale sans cache qui applique une métrique simple pour définir les temps d'exécution des blocs de base (colonnes 3 et 4). Enfin, nous avons analysé les programmes en utilisant un modèle d'architecture dérivé d'ARM9 avec un cache d'instruction de 1 kilo-octet et un cache de données de 256 kilo-octets (colonnes 5 et 6).

Pour chaque architecture, nous procédons à trois analyses de WCET : la première correspond à une estimation de WCET sans inclure les conflits découverts par PathFinder, et sert de base à la comparaison de l'efficacité des deux méthodes d'intégration. La seconde consiste à intégrer les conflits dans l'estimation de WCET au moyen de contraintes ILP additionnelles mentionnée précédemment. La dernière repose sur la méthode de dépliage de graphe à partir des conflits partiels présentée dans la partie 7.2.5. Les colonnes titrées "Contraintes" correspondent au pourcentage de gain de précision du WCET entre la première analyse et la seconde, et les colonnes titrées "Dépliage" correspondent à ce gain de précision entre la première analyse et la dernière.

Par ailleurs, face au nombre parfois important de conflits détectés et la taille de certains de ces conflits, certains programmes n'ont pas pu être analysés en temps raisonnable par la méthode du dépliage. Dans ces cas, la case du tableau comporte la mention *ED* pour *Échec de Dépliage*.

On observer pour commencer des gains significatifs de la précision du WCET lorsque les conflits sont intégrés dans l'analyse, et ce même pour une architecture triviale sans cache. Pour les programmes qui présentent des gains dans les colonnes 3 et 4, l'intégration des conflits a donc permis de préciser le WCEP estimé du programme qui correspondait, avant cette intégration, à un chemin infaisable. De plus, les gains pour les deux méthodes d'intégration sont strictement identiques pour cette architecture triviale, ce qui n'est pas surprenant puisqu'en l'absence de caches et de modélisation par interprétation abstraite de la micro-architecture, on ne peut espérer les types de gains obtenus dans le chapitre précédent.

Par ailleurs, sur l'architecture plus complexe dont les résultats sont présentés dans les colonnes 5 et 6, on observe des différences entre les résultats des deux méthodes (en gras). Ces résultats confirment ce qui a déjà été observé dans le chapitre 6, à savoir qu'il est possible de gagner en précision en intégrant des conflits par le dépliage du graphe plutôt que par des contraintes ILP additionnelles. Les raisons de ce gain sont que les analyses sur la modélisation par interprétation abstraite de la micro-architecture bénéficient de ce dépliage de graphe.

7.4 Conclusion

Ce chapitre a présenté un nouveau type d'annotation appelée *conflits*. Les résultats de l'analyse de l'outil PathFinder ont motivé la spécification d'une telle classe de chemins infaisables. La notion de conflit consiste à indiquer qu'un certains nombre d'éléments du CFG ne peuvent pas tous apparaître dans une même exécution.

Nous avons défini des balises spécifiques dans le langage FFX pour supporter cette annotation et nous nous sommes intéressés à la génération du FFA correspondant. Nous avons cependant fait face à de sévères problèmes de complexité et nous avons cherché un moyen d'affaiblir la notion de conflit de façon à réduire la taille du FFA initial et ses répercutions sur le CFG déplié.

Pour y parvenir, nous avons dérivé l'ordre des arcs d'un conflit de la structure du CFG, ce qui nous a permis d'exprimer un conflit partiel, dont le nombre d'états ne croit pas de façon exponentielle, à la place de l'annotation initiale.

Pour finir, nous avons comparé l'intégration de ces conflits dans une analyse de WCET par la méthode de dépliage et une seconde méthode d'intégration par des contraintes ILP sur plusieurs architectures. Ces expériences nous ont permis de confirmer les résultats du chapitre précédent qui montraient des gains de précision provenant des analyses de la micro-architecture par interprétation abstraite.

Nous avons cependant montré que les problèmes de complexité de l'approche par dépliage n'étaient pas complètement réglés et souffraient d'un nombre trop important de conflits, ou de la présence de conflits impliquant un trop grand nombre d'arcs.

"I'll leave tomorrow's problems to tomorrow's me."

— Saitama

8 Conclusion

Les travaux de recherche présentés dans ce mémoire de thèse s'intègrent dans le domaine de l'analyse statique qui consiste à dériver des propriétés d'un programme sans l'exécuter, et plus particulièrement dans l'estimation du temps d'exécution au pire cas (WCET) des programmes par la méthode d'énumération implicite de chemins (IPET). Cette approche repose sur la représentation sous forme de graphe de flot de contrôle (CFG) des exécutions structurellement possibles d'un programme, associées à un ensemble de propriétés dérivées de la sémantique du code source. Ces propriétés permettent de préciser parmi les exécutions qui respectent la structure du programme, celles qui sont en réalité infaisables, afin qu'elles ne soient pas prises en compte lors de l'estimation de WCET à proprement parler.

Le rôle des langages d'annotation est de supporter ce type de propriétés et d'être capable de les fournir à l'outil d'analyse en charge de l'estimation de WCET. Dans la méthode IPET, le processus d'intégration de ces annotations dans l'analyse consiste à générer des ensembles de contraintes numériques portant sur les composants du CFG. À terme, l'ensemble des informations sur le programme, incluant la structure du CFG en elle-même, sera combiné avec les durées d'exécution de ses instructions, issues d'une

modélisation de l'architecture matérielle, dans un problème d'optimisation linéaire en nombres entier (ILP) dont la résolution fournira une borne du WCET du programme.

Nous présentons dans cette thèse un nouveau formalisme d'automates permettant l'expression des différentes annotations habituellement utilisées dans le domaine de l'analyse de WCET. Ces automates enrichis de contraintes, de variables, et d'une hiérarchie sont capables de représenter, à partir d'annotations fournies par un expert ou un outil, des ensembles de chemins faisables dans le CFG du programme, et par complément d'interdire des chemins infaisables. Le processus d'intégration des annotations portées par les automates dans l'analyse de WCET repose sur l'intersection des chemins structurels autorisés représentés par le CFG et ceux autorisés par les automates.

Les fonctionnalités de ces automates appelés automates d'annotation de flot ont été pensées et définies précisément pour supporter des annotations de flot à destination de la méthode IPET. D'un côté, des contraintes globales associées à des variables attachés aux arcs des automates permettent d'exprimer les annotations usuelles comme les bornes de boucles du programme, des contraintes sur le nombre d'exécutions d'éléments spécifiques du CFG, ou des relations numériques entre certains de ces éléments. Ces contraintes et variables supportent ainsi aisément la méthode traditionnelle d'intégration d'annotations par l'enrichissement du système de contraintes destiné à être résolu par ILP. De l'autre, une hiérarchie d'automate permet de représenter les différents contextes d'exécution usuels de la programmation impérative, comme les contextes de boucle, de fonction, d'itération, etc. La combinaison de ces deux fonctionnalités offre un support d'annotations de flot contextuelles équivalent aux langages d'annotation les plus expressifs.

L'intégration des annotations à partir de ces automates dans le CFG trouve ses origines dans le produit d'automates de la littérature. Il a cependant du être adapté en raison de la complexité des automates hiérarchiques à contraintes présentés et formalisés dans ce mémoire. Ainsi, le processus d'intégration est divisé en deux étapes majeures qui sont l'injection d'un automate équivalent au CFG du programme dans un automate hiérarchique à contraintes généré à partir d'annotations de flot, et l'aplatissement du résultat de cette injection permettant la reconstruction du CFG.

Nous avons présenté les algorithmes correspondant à ces opérations et nous avons montré comment il était possible de générer des automates porteurs d'annotations contextuelles à partir du langage d'annotation FFX. En parallèle, nous avons implémenté

ces automates enrichis et les opérations nécessaires sous forme de plug-in dans l'outil d'analyse statique académique Otawa. Nous étions ainsi en mesure de vérifier l'efficacité et la valeur de notre approche.

Par ailleurs, à partir de l'expressivité de nos automates enrichis, nous avons mis en évidence la possibilité de représenter diverses annotations dans la structure même des automates, en remplacement des contraintes. Nous avons montré sur des exemples que l'intégration automatique, utilisant les opérations d'injection et d'aplatissement, d'annotations sous forme d'automates dépliés provoque un dépliage partiel du CFG, ce qui a confirmé la réussite de cette intégration d'annotations.

Les automates hiérarchiques à contraintes nous permettent donc d'intégrer certaines annotations de deux façons, soit en utilisant l'expressivité des contraintes et des variables, soit en dépliant les états de l'automate.

Nous avons comparé ces deux méthodes sur un certain nombre de programmes, et nous avons observé des gains de précision du WCET estimé par la méthode de dépliage de graphe. Nous avons interprété et expliqué ces différences par les effets du dépliage sur certaines analyses relatives à la micro-architecture.

Additionnellement, un type d'annotations spécifique issu d'un outil de détection de chemins infaisables a motivé la définition d'une classe spécifique d'annotation dans le langage FFX, ainsi que sa traduction en automates. La possibilité d'expression de ces annotations par des automates dépliés est également apparue, accompagnée cependant d'importants problèmes de complexité. Nous avons dérivé une propriété d'ordre entre les arcs du CFG afin de surmonter ces problèmes, et de nouveaux résultats expérimentaux nous ont permis de confirmer la tendance de la méthode de dépliage à être plus précise que la méthode traditionnelle.

Les différents travaux effectués ont donc permis de définir une alternative efficace aux langages d'annotation et dont l'expressivité offre de nouvelles possibilités d'intégration des annotations, avec des bénéfices potentiels en terme de précision de l'estimation du WCET.

Tous les problèmes de complexité ne sont pas résolus pour autant, et des travaux restent également à faire sur l'efficacité du plug-in développé, notamment en terme de temps d'exécution de l'analyse. Par ailleurs, nous avons entrevu des dépliages de contextes spécifiques (la dernière itération d'un boucle par exemple) impliquant de l'indéterminisme dans les automates construits ainsi que dans le CFG. Ajouter

de l'indéterminisme aux automates hiérarchiques à contraintes permettrait effectivement de supporter des types de contextes et de contraintes additionnelles et semble être une piste prometteuse pour étendre ces travaux. Enfin, dans les divers résultats expérimentaux montrant un gain de précision avec la méthode de dépliage, nous nous sommes penchés en détail sur les effets de ce dépliage sur la modélisation du cache. La modélisation des autres éléments de la micro-architecture pourraient également être impactés par ce dépliage et mériteraient d'être étudiés en ce sens.

Annexes

ANNEXE A

Grammaire FFX partielle

```
FUNCTION ::=
  <function LOCATION−ATTRS
        INFORMATION−ATTRS>
    STATEMENT+
  </function>

STATEMENT ::=
| LOOP
| CALL
| CONTROL−CONSTRAINTS
| ...

BLOCK ::=
  <block LOCATION−ATTRS
    INFORMATION−ATTRS/>

LOOP ::=
  <loop LOCATION−ATTRS
    INFORMATION−ATTRS>
  </loop>
| <loop LOCATION−ATTRS
    INFORMATION−ATTRS>
    <iteration number="INT">
    </iteration>
  </loop>

CALL ::=
  <call LOCATION−ATTRS
    INFORMATION−ATTRS>
    FUNCTION+
  </call>

<!−− location attributes −−>
LOCATION−ATTRS ::=
| IDENTIFICATION
| ADDRESS−LOCATION
| LABEL−LOCATION
| SOURCE−LOCATION

IDENTIFICATION ::= id = "TEXT"

ADDRESS−LOCATION ::=
  <element address="INT"/>

SOURCE−LOCATION ::=
| <element source="TEXT" line="INT"/>
| <element line="INT"/>

<!−− loop bound attributes −−>
LOOP−ATTR ::=
| maxcount="INT|NOCOMP"?
| totalcount="INT|NOCOMP"?
| ...

expression ::= INT
| '−' expression
| '(' expression ')'
| expression '+' expression
| expression '−' expression
| expression '*' expression
| expression '/' expression

<!−− control flow constraints −−>
CONTROL−CONSTRAINT ::=
  <control−constraint> CONTROL−PREDICATE
  </control−constraint>

CONTROL−PREDICATE ::=
  | CONTROL−RELATION
  | <and> CONTROL−RELATION+ </and>
  | <or> CONTROL−RELATION+ </or>

CONTROL−RELATION ::=
  | <eq> CONTROL−FORMULA CONTROL−FORMULA
    </eq>
  | <ne> CONTROL−FORMULA CONTROL−FORMULA
    </ne>
  | <lt> CONTROL−FORMULA CONTROL−FORMULA
    </lt>
  | <le> CONTROL−FORMULA CONTROL−FORMULA
    </le>
  | <gt> CONTROL−FORMULA CONTROL−FORMULA
    </gt>
  | <ge> CONTROL−FORMULA CONTROL−FORMULA
    </ge>

CONTROL−FORMULA ::=
  | <int>INT</int>
  | <count ref="TEXT"/>
  | <neg> CONTROL−FORMULA </neg>
  | <add> CONTROL−FORMULA CONTROL−FORMULA
    </add>
  | <sub> CONTROL−FORMULA CONTROL−FORMULA
    </sub>
  | <mul> CONTROL−FORMULA CONTROL−FORMULA
    </mul>
  | <div> CONTROL−FORMULA CONTROL−FORMULA
    </div>
```

Acronymes

CFA automate de flot de contrôle ou *Control Flow Automaton*. 25

CFG graphe de flot de contrôle ou *Control Flow Graph*. 7

DFA automate fini déterministe ou *Deterministic Finite Automaton*. 33

FFA automate d'annotation de flot ou *Flow Fact Automaton*. 27

FFX *Flow Fact in XML*. 18

HFA automate d'annotation de flot hiérarchique ou *Hierarchical Flow fact Automaton*. 51

ILP programmation linéaire en nombres entiers ou *Integer Linear Programming*. 17

IPET méthode d'énumération implicite de chemins ou *Implicit Path Enumeration Technique*. 10

WCEP chemin d'exécution au pire cas ou *Worst-Case Execution Path*. 13

WCET temps d'exécution au pire cas ou *Worst-Case Execution Time*. 6

XML langage de balisage extensible ou *Extensible Markup Language*. 18

Bibliographie

[1] WCET Tool Challenge. http ://www.mrtc.mdh.se/projects/WTC/.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7) :1–19, July 1970.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA : an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[5] Jean-Luc Béchennec and Franck Cassez. Computation of WCET using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011.

[6] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[7] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer, 1984.

[8] Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX : a portable WCET annotation language. In Liliana Cucu-Grosjean, Nicolas Navet, Christine Rochange, and James H. Anderson, editors, *20th International Conference on Real-Time and Network Systems, RTNS '12, Pont a Mousson, France - November 08 - 09, 2012*, pages 91–100. ACM, 2012.

[9] Alexandre Boudet and Hubert Comon. Diophantine equations, presburger arithmetic and finite automata. In Hélène Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1996.

[10] Franck Cassez and Jean-Luc Béchennec. Timing analysis of binary programs with UPPAAL. In Josep Carmona, Mihai T. Lazarescu, and Marta Pietkiewicz-Koutny, editors, *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 41–50. IEEE Computer Society, 2013.

[11] Vasek Chvatal. *Linear programming.* Macmillan, 1983.

[12] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 37–44. IEEE, 2001.

[13] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In Hu and Vardi [28], pages 268–279.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[15] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC : modular execution time analysis using model checking. In Björn Lisper, editor, *10th International Workshop on*

Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium, volume 15 of *OASICS*, pages 113–123. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[16] Jakob Engblom. *Processor pipelines and static worst-case execution time analysis*. PhD thesis, 2002.

[17] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*, pages 88–95. IEEE Computer Society, 1999.

[18] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*, pages 163–174. IEEE Computer Society, 2000.

[19] Andreas Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, 2003.

[20] Christian Ferdinand, Reinhold Heckmann, Henrik Theiling, and Reinhard Wilhelm. Convenient user annotations for a WCET tool. In Jan Gustafsson, editor, *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, volume MDH-MRTC-116/2003-1-SE, pages 17–20. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.

[21] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9) :1563–1581, 1966.

[22] Jan Gustafsson. *Analyzing execution-time of object-oriented programs using abstract interpretation*. PhD thesis, 2000.

[23] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks : Past, present and future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[24] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005), 2-4 February 2005, Sedona, AZ, USA*, pages 287–300. IEEE Computer Society, 2005.

[25] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of c-programs for WCET calculation. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*, pages 106–112. IEEE Computer Society, 2003.

[26] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[28] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[29] Raimund Kirner. The programming language wcetc. *Technical report, Technische Universit at Wien, Institut fur Technische Informatik*, 2002.

[30] Raimund Kirner. *Extending optimising compilation to support worst-case execution time analysis*. PhD thesis, 2003.

[31] Raimund Kirner. The WCET analysis tool calcwcet167. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012.

[32] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds : comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 10(3) :411–437, 2011.

[33] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET analysis : The annotation language challenge. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 6 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[34] Raimund Kirner and Peter P. Puschner. Timing analysis of optimised code. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*, pages 100–105. IEEE Computer Society, 2003.

[35] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies. (AM-34) (Annals of Mathematics Studies)* [60].

[36] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. r-tubound : Loop bounds for WCET analysis (tool paper). In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 435–444. Springer, 2012.

[37] Hanbing Li. *Extraction and traceability of annotations for WCET estimation. (Extraction et traçabilité d'annotations pour l'estimation de WCET)*. PhD thesis, University of Rennes 1, France, 2015.

[38] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995.

[39] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings*, pages 298–307. IEEE Computer Society, 1995.

[40] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software : beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, pages 254–263. IEEE Computer Society, 1996.

[41] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Sym-*

posium, Phoenix, AZ, USA, December 1-3, 1999, pages 12–21. IEEE Computer Society, 1999.

[42] Alexander Metzner. Why model checking can improve WCET analysis. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer, 2004.

[43] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008.

[44] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008.

[45] Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne de Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in WCET analysis. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2016.

[46] Vincent Mussot and Pascal Sotin. Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 207–216. IEEE Computer Society, 2015.

[47] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. In Richard Gerber and Thomas J. Marlowe, editors, *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995). La Jolla, California, June 21-22, 1995*, pages 20–30. ACM, 1995.

[48] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 72–81. IEEE Computer Society, 1990.

[49] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*, page 442. IEEE Computer Society, 1999.

[50] Peter Puschner. The single-path approach towards wcet-analysable software. In *Industrial Technology, 2003 IEEE International Conference on*, volume 2, pages 699–704. IEEE, 2003.

[51] Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2) :159–176, 1989.

[52] Peter P. Puschner and Roman Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 134–143. IEEE Computer Society, 1998.

[53] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 13(1) :67–91, 1997.

[54] UK Rapita Systems Ltd., York. RapiTime toolkit for dynamic analysis, 2006.

[55] Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In Tulika Mitra and Jan Reineke, editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 8 :1–8 :9. ACM, 2014.

[56] Jordy Ruiz and Hugues Cassé. Using SMT solving for the lookup of infeasible paths in binary programs. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASICS*, pages 95–104. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[57] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In Y. Annie Liu and Reinhard Wilhelm, editors, *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99), Atlanta, Georgia, USA, May 5, 1999*, pages 35–44. ACM, 1999.

[58] Boston Scientific. Pacemaker system specification. *Boston Scientific*, 2007.

[59] Daniel Sehlberg, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Steffen Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 212–219. IEEE, 2006.

[60] C. E. Shannon and J. McCarthy. *Automata Studies. (AM-34) (Annals of Mathematics Studies)*. Princeton University Press, Princeton, NJ, USA, 1956.

[61] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Software Eng.*, 15(7) :875–889, 1989.

[62] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4) :339–355, 2000.

[63] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 144–153. IEEE Computer Society, 1998.

[64] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3) :157–179, 2000.

[65] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, 2005.

[66] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1) :230–265, 1937.

[67] Leslie G. Valiant and Mike Paterson. Deterministic one-counter automata. *J. Comput. Syst. Sci.*, 10(3) :340–350, 1975.

[68] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 606–611. IEEE Computer Society, 2005.

[69] Reinhard Wilhelm. Why AI + ILP is good for wcet, but MC is not, nor ILP alone. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 309–322. Springer, 2004.

[70] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.