Projet RNTL AVERILES
Fourniture F2.2 : Algorithmes de Vérification
Rapport final

LIAFA, CRIL, EDF, LSV, VERIMAG

revue de fin de projet le 6 novembre 2009

Introduction

Dans le cadre du projet RNTL AVERILES, des algorithmes de vérification symboliques ont été développés par les différents partenaires académiques. Ces algorithmes permettent de vérifier des programmes mono-tâche ou multi-tâches utilisant des pointeurs, des entiers, et des verrous. La plupart de ces algorithmes ont de plus été implémentés dans les différents outils développés au sein du projet ; certains de ces outils font partie des livrables du projet, d'autres sont encore en cours de développement et sont présentés ici à titre informatif.

Le but de ce rapport est de donner une idée générale du fonctionnement des différents algorithmes, les détails plus techniques étant disponibles dans les publications associées, dont la liste est fournie en bibliographie.

La présentation des algorithmes est donc un guide de lecture des articles réalisés dans le projet ainsi qu'un inventaire des techniques issues du model-checking utilisées dans les outils ou dans certains résultats théoriques. On a distingué trois grandes catégories d'algorithmes :

- les algorithmes des outils de vérification de programmes mono-tâche,
- les algorithmes des outils de vérification de programmes multi-tâches,
- les algorithmes qui n'ont pas été implémentés.

Ces trois catégories correspondent aux trois parties du rapport.

Une approche basée sur le model-checking

Précisons que certains aspects informatiques sont idéalisés : les entiers sont des entiers mathématiques, la mémoire allouable est potentiellement infinie, etc. Dans certains cas, les tableaux peuvent même couvrir une zone mémoire infinie, et on omet les dépassements de bornes. Enfin, les appels de fonctions sont en général gérés de manière limitée (pile d'appel bornée) et font l'objet d'un pré-traitement avant d'appliquer l'algorithme. Ces algorithmes sont donc à prendre dans une perspective de vérification de la correction fonctionnelle des programmes et des algorithmes qu'ils implémentent.

Cette approche propre au model-checking est orthogonale et complémentaire de celle de l'analyse statique, par interprétation abstraite notamment, qui modélise bien plus finement la sémantique réelle des langages de programmation mais ne vérifie pas la correction fonctionnelle de programmes implémentant des algorithmes élaborés.

Cette approche est aussi à comparer à la preuve formelle de programmes, manuelle ou semi-automatique, dont l'objet d'étude est la plupart du temps des programmes courts au principe de fonctionnement complexe.

Notre approche est un premier pas vers un objectif difficile, à savoir la mise en commun du meilleur des deux autres approches : des outils entière-

ment automatiques, et des cas d'études dont les principes de fonctionnement sont complexes.

Liste des algorithmes

Le tableau de la figure 1 résume l'ensemble des algorithmes, des outils, et des types de programmes ou de modèles mémoires analysés. Concernant les types de programmes, précisons brièvement les caractéristiques prises en compte :

- **nombre de sélecteurs :** les structures récursives contiennent un certain nombre de champs, certains étant des pointeurs vers d'autres structures. Ce sont ces champs que l'on compte comme sélecteurs. Une liste pure et une liste d'entier n'ont qu'un seul sélecteur, un arbre et une liste doublement chaînée ont deux sélecteurs, une liste doublement chaînée avec pointeur de tête a trois sélecteurs, etc.
- **données :** on précise ici quelle est la prise en compte des données manipulées par le programme : les structures de données récursives ou les tableaux contiennent en général des données (entiers, flottants,...) dont les propriétés d'égalité, d'ordonnancement, de rapport arithmétique peuvent être énoncés.
- **compteurs :** on précise si l'algorithme prend en compte les variables entières.
- **verrous :** on précise si les verrous sont alloués statiquement, et donc en nombre fixé par avance dans le programme, ou si ils sont alloués dynamiquement (pour les algorithmes de listes concurrentes à faible exclusion mutuelle, notamment).
- **arithmétique de pointeurs :** on précise si l'algorithme tient compte de l'arithmétique de pointeurs, notamment de la possibilité de représenter des tableaux.

Algo.	**Outil**	**nbr. sel.**	**données**	**cmpt.**	**verrous**	**arith. ptr.**
formes & compteurs	L2Ca Topics	1	non	oui	non	non
autom. arbres	Artmc	∞	non	non	non	non
sys. à compteurs	exper.	0	entiers (arith.)	oui	non	oui
produit. autom.	Topics	1	non	non	statiques	non
formes étendues	exper.	1	entiers (ordre)	oui	dyn.	non
logique de séparation	L2Ca	∞	abstrait (ordre)	non	non	oui
logique CSL	non	∞	param. (théorie dec.)	oui	non	oui
chgt. ctxt. borné	non	0	abstrait. (égalité.)	non	statiques	non

Fig. 1 – Liste des algorithmes développés dans le projet Averiles (ordre de présentation dans ce rapport)

Table des matières

Chapitre 1

Programmes simple tâche

1.1 Analyse de formes avec compteurs

1.1.1 Programmes et propriétés étudiées

Dans [16] et [17], les auteurs proposent une méthode pour analyser les programmes manipulant des structures de type listes simplement chaînées pouvant être décrites comme indiqué par la figure 1.1 (selon la syntaxe du C). Le champ `next` pointe vers la cellule suivante dans la liste simplement chaînée

```
typedef struct node
{
 struct node *next;
 int data;
} *List
```

FIG. 1.1 – Structure de liste utilisée

et le champ `data` pointe vers la donnée contenue dans la cellule. Dans [16], les programmes considérés ne contiennent pas d'appel au champ `data` (ceux-ci sont donc abstraits du programme original), en revanche dans [17], il est possible de considérer les données incluses dans les cellules afin de pouvoir parler de listes ordonnées. Les programmes considérés ne sont pas concurrents et ne contiennent pas d'appels récursifs de fonctions, ils peuvent ainsi être considérés comme des procédures travaillant sur des variables globales.

Les variables peuvent être soit des variables de pointeurs, soit des variables entières. Sur les variables de pointeurs, les instructions suivantes peuvent être réalisées : les mises à jour de variables tels que `u :=null`, `u :=w` ou `u :=w.next`, les mises à jours de sélecteurs `u.next=null` ou `u.next :=w`, les créations `u :=new` et les destructions de cellules `free(u)`. En ce qui concerne les variables entières, les instructions considérées sont l'incrémentation `i :=i+1`, la décrémentation `i :=i-1` et la mise à zéro `i :=0`. Il est

également possible de réaliser des tests sur ces variables tels que l'égalité de pointeurs `u=w` ou `u=null` et le test à zéro sur les entiers `i=0`. Les programmes sont alors représentés sous forme d'automates finis étendus où chaque transition est étiquetée par un test booléen et une instruction.

Les propriétés que l'on cherche à vérifier sur ces programmes sont, dans un premier temps, l'absence de violation mémoire et l'absence de fuite mémoire, mais il s'avère que l'algorithme proposé permet aussi de vérifier des propriétés plus complexes sur la forme du tas mémoire (par exemple qu'un programme ne crée jamais de liste cyclique) et sur le nombre de cellules manipulées (par exemple deux listes ont toujours la même longueur). Dans certains cas, cet algorithme permet également de vérifier la terminaison du programme.

1.1.2 Représentation symbolique de la mémoire

Le tas mémoire est représenté sous forme d'un graphe (appelé graphe mémoire) où chaque noeud a un unique successeur et peut être étiqueté par une variable de pointeurs. Il existe aussi un noeud spécial appelé `null` (correspondant à l'adresse mémoire `NULL`). Les différentes instructions du programme modifient alors soit la forme du graphe, soit la position des variables sur les noeuds du graphe.

Dans [22], une représentation abstraite de tels graphes a été proposée. Les graphes mémoire abstraits consistent en des graphes dans lesquels chaque noeud est soit pointé par au moins deux noeuds, soit étiqueté par une variable. On associe de plus à chaque noeud du graphe mémoire abstrait une variable entière. Une propriété intéressante de ces graphes mémoire abstraits est que pour un nombre fixé de variables de pointeurs, il n'existe qu'un nombre fini de graphes mémoire abstraits (modulo isomorphisme de graphes et renommage des variables entières). De plus en associant une valeur (strictement positive) à chaque variable entière, il est possible d'obtenir un unique graphe mémoire, la valeur donnant le nombre de successeurs du noeud auquel la variable correspondante est associée.

1.1.3 Algorithme de vérification

L'algorithme de vérification proposé repose sur une traduction des programmes vers un système à compteurs bisimilaire. Un système à compteurs est un automate fini étendu manipulant uniquement des variables entières et dont les transitions sont étiquetées par un test (combinaison booléenne de tests à zéro sur les variables entières) et par une opération sur les différentes variables entières (incrémentation ou décrémentation). Notons, qu'en réalité, la syntaxe des opérations sur les variables entières peut être étendue à des fonctions linéaires de façon à avoir un automate plus concis. Les variables entières du système à compteurs correspondent ainsi aux variables entières

du programme et aux variables entières présentes sur les différents graphes mémoire abstraits.

La vérification des propriétés telles que l'absence de violation mémoire ou l'absence de fuite mémoire sur le programme se ramène alors à un problème d'accessibilité d'un état de contrôle sur le système à compteurs obtenu. Remarquons que bien que ce dernier problème soit dans le cas général indécidable, il existe des outils tels que FAST [21, 23] qui permettent d'analyser les systèmes à compteurs. Bien que FAST soit basé sur un semi-algorithme et qu'il se puisse que son analyse ne termine pas, dans la pratique, ce model-checker a permis d'analyser un grand nombre de cas d'études.

L'algorithme de vérification se décompose donc en deux étapes :

1. Traduire le programme dans un système à compteurs
2. Analyser le système à compteurs obtenu avec un model-checker de systèmes à compteurs comme FAST

Un point important est que la construction du système à compteurs bisimilaire est toujours possible et lors de cette phase il est parfois possible de vérifier l'absence d'erreurs sans analyse ultérieure.

1.1.4 Résultats obtenus

La traduction vers des systèmes à compteurs a permis d'analyser la plupart des programmes classiques travaillant sur les listes simplement chaînées comme les fonctions `reverse`, `delete`, `merge`, etc. De plus, les outils L2CA et TOPICS développés dans le cadre du projet AVERILES utilisent cet algorithme de traduction.

1.2 Model-checking régulier abstrait

Dans le cadre du projet nous avons poursuivi [15, 7, 5] des travaux basés sur le model-checking régulier abstrait [27, 25]. Ces travaux ont permis d'obtenir des résultats sur les programmes avec structures de mémoire dynamique en forme de *listes simplement chaînées*. Dans [18] nous avons étendu ces résultats pour traiter des structures de mémoire plus complexes.

1.2.1 Programmes et propriétés vérifiées

Nous considérons des programmes C non récursifs qui manipulent des structures dynamiques avec *plusieurs* sélecteurs et avec des données sur un domaine fini permettant par exemple de décrire des listes doublement chaînées ou des arbres avec leurs feuilles liées entre elles. Les propriétés vérifiées sont les propriétés de base (pas de déréférencement de pointeur null, etc.) ainsi que des invariants de forme (par exemple à la fin d'un programme on

obtient une liste doublement chaînée, etc.). Les invariants de forme traités sont ceux dont la violation peut être spécifiée dans le fragment existentiel d'une logique du premier ordre sur les graphes que nous avons définie. Cette logique, appelée LBMP (*Logic of Bad Memory Patterns*), permet de décrire de *mauvaises situations* (par exemple qu'à un endroit la liste n'est pas correctement doublement chaînée).

Les propriétés décrites en LBMP peuvent être traduites vers des morceaux de programmes C (appelés testeurs), qui testent ces propriétés et qui vont vers un état d'erreur si une *mauvaise situation* est atteinte. Ainsi, la vérification se réduit à l'accessibilité de l'état d'erreur. Les propriétés peuvent aussi être directement décrites comme des testeurs.

1.2.2 Représentation symbolique de la mémoire

Notre méthode de vérification est basée sur l'approche du model-checking régulier abstrait ou *Abstract Regular Tree Model Checking* (ARTMC) [26]. Dans ARTMC les configurations d'un système sont des arbres sur un alphabet gradué fini, les ensembles de configurations sont décrits par des automates d'arbre et les transitions du système sont données comme des transducteurs d'arbre. Ensuite, l'ensemble des configurations atteignables à partir d'un ensemble initial est calculé en appliquant d'une façon répétitive le transducteur d'arbre sur les configurations atteignables jusqu'à présent. Pour que ce calcul s'arrête le plus souvent possible, plusieurs abstractions sur les automates d'arbre sont utilisées en ARTMC. Ces abstractions peuvent être automatiquement raffinées si nécessaire.

Pour pouvoir appliquer cette méthode dans notre cadre, nous définissons un codage des configurations de mémoire (qui sont des graphes généraux) comme des arbres. Un graphe mémoire est représenté par un squelette (un arbre) et les arêtes qui ne sont pas présentes dans ce squelette sont données par des *expressions de routage* sur le squelette. Ces expressions sont des expressions régulières sur les directions de l'arbre (par exemple gauche, droite, haut, etc.) et elles indiquent les extrémités possibles des arêtes représentées. Par exemple, une liste doublement chaînée peut être représentée par une liste simple (qui est un arbre) *et* des arêtes supplémentaires qui décrivent le fait que chaque cellule (mise à part la première) a aussi un pointeur qui pointe vers son prédécesseur.

Pour toutes les opérations de manipulation de pointeurs utilisées en C, nous définissons des transducteurs d'arbre correspondants. Nous pouvons ainsi appliquer ARTMC.

1.2.3 Algorithme de vérification

Nous utilisons la méthode d'ARTMC décrite ci-dessus pour la vérification qui consiste à montrer qu'un état d'erreur n'est pas atteignable.

1.2.4 Résultats obtenus

Nous avons appliqué notre méthode à plusieurs études de cas : listes doublement chaînées, arbres (notamment l'algorithme Deutsch-Schorr-Waite), listes de listes, et arbre avec liens transverses entre feuilles.

1.3 Programmes à tableaux d'entiers

1.3.1 Programmes et propriétés vérifiées

Nous avons étudié les programmes travaillant avec des tableaux d'entiers, de taille a priori non bornée. Le but est, étant donnée une précondition et une post-condition écrite dans la logique SIL [10, 11] de vérifier que le triplet de Hoare ainsi défini est valide. Le programme peut contenir des boucles (non emboitées), et les invariants de boucle n'ont pas besoin d'êtres précisés. La logique SIL (pour Single Index Logic) permet ainsi de spécifier des propriétés sur une famille de tableaux en utilisant un parcours de ces tableaux n'utilisant qu'un seul index. Les programmes traités sont donc uniquement les programmes qui n'utilisent qu'un seul index de parcours dans chaque boucle, par exemple un programme qui recopie un tableau dans un autre, incrémente un tableau case par case, etc.

1.3.2 Représentation symbolique de la mémoire

La représentation mémoire utilisée dans cette approche [3] se base sur le fait qu'on peut voir un ensemble de configurations d'une famille de tableaux comme le langage d'un automate à compteurs, une configuration des tableaux formant un mot de vecteurs d'entiers, de dimension le nombre de tableaux considérés. On peut de même représenter les instructions de modification de tableaux au sein de boucles comme des transducteurs de mots de vecteurs d'entiers.

1.3.3 Algorithme de vérification

La partie la plus sensible de l'algorithme consiste à traduire les formules de SIL en automates à compteurs, et réciproquement les automates à compteurs en formule de SIL. Ceci n'est possible qu'en posant certaines restrictions sur les automates à compteurs considérés, notamment la platitude de la structure de controle (pas de boucles imbriquées).

1.3.4 Résultats obtenus

Nous avons validé notre approche en vérifiant avec succès [3] de manière semi-automatique des triples de Hoare dans la logique SIL, notamment un programme de séparation de tableau en partie positive et négative, la

rotation de tableau, l'insertion dans un tableau trié, etc. Certaines étapes doivent nécessitent encore une assistance humaine, car toutes les techniques envisagées n'ont pas encore été implémentées.

Chapitre 2

Programmes multi-tâches

2.1 Programmes à verrous statiques

2.1.1 Présentation et hypothèses

Après avoir proposé différentes approches pour analyser le comportement de programmes mono-tâche, nous avons étudié dans quelle mesure ces algorithmes de vérification pouvaient permettre d'analyser des programmes multi-tâches. Les programmes que nous considérons sont constitués de plusieurs tâches qui s'exécutent en parallèle, chacune de ces tâches manipulant une mémoire globale tout comme une mémoire locale. Nous ajoutons de plus des verrous de façon à ce que les différentes tâches puissent se synchroniser et aussi protéger leur accès à la mémoire.

En plus des instructions considérées pour les programmes mono-tâche, les programmes multi-tâches peuvent lancer des tâches avec une instruction de la forme `pthread_create` qui prend en argument la fonction que la nouvelle tâche exécutera, une tâche peut également attendre que l'une des tâches qu'elle a créée au préalable ait fini son exécution grâce à une instruction de la forme `pthread_join`. En ce qui concerne la manipulation des verrous, il est possible de créer un verrou (`pthread_mutex_init`), de détruire un verrou (`pthread_mutex_destroy`), de prendre un verrou (`pthread_mutex_lock`), de libérer un verrou (`pthread_mutex_unlock`) et de tester si un verrou est libre (`pthread_mutex_trylock`). Parmi ces instructions, `pthread_join` et `pthread_mutex_lock` sont bloquantes.

Pour l'analyse de tels programmes multi-tâches, deux critères peuvent alors être pris en compte :

1. le nombre de tâches lancées est borné ou non
2. la création des verrous se fait de façon statique ou dynamique

Dans le cadre de ce projet, l'analyse se portera uniquement sur des programmes ayant un nombre de tâches lancées fini, nous supposons ainsi qu'il y a une tâche principale dont le rôle est de lancer les autres et dont le code ne comporte aucune boucle ni aucun appel de fonctions. En ce qui concerne les autres tâches elles ne peuvent pas lancer elle-mêmes de tâches. Notons que la tâche principale peut aussi initialiser certaines données. Dans cette partie, nous présentons une méthode pour analyser les programmes multi-tâches lorsque les verrous sont statiques, nous supposons donc que les verrous sont des variables globales, qu'ils sont créés (et éventuellement détruits) par la tâche principale et les seules actions qui peuvent être réalisées par les autres tâches sont la prise du verrou, la libération du verrou ainsi que tester si un verrou est libre ou non.

2.1.2 Produit d'automates et suppression des verrous

Les programmes mono-tâche pour lesquels nous avons proposé des méthodes d'analyse dans le chapitre précédent peuvent être vues comme des automates finis dont chacune des transitions est étiquetée par un test et une instruction, cette vision permet en particulier de capturer l'aspect non-déterminisme de ces programmes qui peut apparaître suite à une phase d'extraction de modèles à partir d'un programme réel. Avec les hypothèses que nous avons posées, un programmes multi-tâches peut être ainsi vu comme un ensemble fini d'automates, chaque automate caractérisant le comportement d'une des tâches (ceci étant possible car le nombre de tâches des programmes considérés est fini). Une première étape consiste à passer d'une représentation dans laquelle nous avons un ensemble d'automates à une représentation avec un seul automate. Pour cela, nous réalisons une opération classique sur les automates qui est le produit d'automates, et qui consiste à construire un unique automate pour lequel chaque état contient un état de chaque automate considéré dans le produit. Nous ne détaillons pas plus ici cette opération qui est assez classique dans le cadre de l'analyse de systèmes concurrents.

Une fois cette étape réalisée, nous obtenons un unique automate utilisant des instructions sur les verrous. De façon à pouvoir appliquer les méthodes de vérification pour les programmes mono-tâche, nous souhaitons éliminer ces instructions. L'idée que nous utilisons peut alors être résumée de la façon suivante. Nous encodons dans chaque état de contrôle de l'automate le statut de chacun des verrous. Ce statut peut être représenté par un entier qui vaut soit 0 si le verrou est libre, soit n si le verrou est pris par la tâche n (nous associons un identifiant entier à chacune des tâches). Ainsi lorsque par exemple, la transition sortant d'un état de contrôle est étiquetée par une instruction réalisant la prise d'un verrou, si le statut du verrou correspondant vaut 0, nous mettons à jour ce statut dans l'état d'arrivée de la transition et nous supprimons l'instruction de la transition (sans bien entendu supprimer

la transition), si en revanche le statut du verrou est différent de 0, cela signifie qu'une tâche possède déjà le verrou et par conséquent nous supprimons cette transition de l'automate, car nous savons qu'elle ne sera pas exécutée. Remarquons que cette technique d'encodage du statut de chacun des verrous dans les états de contrôle de l'automate est possible car nous avons un nombre fini de tâches et de verrous.

2.1.3 Remarques et expériences

Nous présentons dans cette section une méthode qui nous permet d'analyser une certaine classe de programmes multi-tâches manipulant dynamiquement la mémoire en réutilisant les méthodes déjà développées pour le cas mono-tâche. Ainsi les opérations réalisant le produit d'automates et la suppression des verrous peuvent être faits dans une phase précédant l'analyse, ce qui permet d'utiliser ensuite les différents outils déjà existants. De plus même si le produit d'automates peut en soit amener à une explosion du nombre d'états, l'analyse des verrous permet de réduire de façon notoire la taille de l'automate finalement analysé. Avec cette approche nous avons pu, grâce aux outils ARTMC, L2CA et TOPICS, analyser des programmes concurrents, comme par exemple un programme mettant en jeu un producteur et un consommateur communiquant via une liste simplement chaînée.

2.2 Programmes à verrous dynamiques

Certains programmes multi-tâches utilisent un nombre non borné de verrous pour se synchroniser, ce qui rend l'approche décrite dans la section précédente impossible à adapter. Dans le projet AVERILES, nous avons entamé l'étude de ces programmes, qui nécessite des techniques parfois très différentes des techniques classiques.

Comme mentionné précédemment, nous n'avons considéré dans ce projet que des programmes multi-tâches à nombre de threads fixé. D'un point de vue théorique, il aurait été nécessaire pour certains programmes de considérer que le nombre de threads n'est pas fixé. Nous n'avons pas cherché à développer ce point, mais nos méthodes tentent de déceler une erreur ou certifient l'absence d'erreur pour un nombre de threads limité. Ceci reste en pratique relativement convaincant, la plupart des bugs connus faisants intervenir en général un faible nombre de threads (très souvent deux threads).

Nous décrivons tout d'abord les spécificités de ces programmes, puis nous détaillons la méthode utilisée, et nous détaillons enfin les résultats obtenus sur un outil prototype.

2.2.1 Programmes à faibles exclusion mutuelle

Les algorithmes concurrents récents, tels ceux qui implémentent les structures de données concurrentes (listes, file, pile, tas, etc ; voir par exemple `java.util.concurrent`) sont souvent basés sur une discipline de programmation qui s'accorde mal avec les méthodes de vérification pour les programmes à nombre de verrous bornés. En effet, ces programmes interfèrent entre eux en général, et tentent d'éviter les synchronisations bloquantes pour libérer tout le gain potentiel de la parallélisation, notamment sur des multicoeurs. Ces algorithmes utilisent tantôt des instructions atomiques, tantôt des verrous *locaux* qui protègent par exemple l'accès à une seule cellule d'une liste partagée. Chaque cellule de la liste contient donc un verrou, ce qui engendre un nombre non borné de verrous. Un exemple connu de tel algorithme est le *lock-coupling* : on parcours une liste en prenant les verrous et en les relâchant au fur et à mesure du parcours. D'autres algorithmes plus efficaces tentent de minimiser le recours aux verrous ou l'utilisation d'instructions atomiques en adoptant une attitude optimiste : les chances d'interférences entre threads sont faibles. Si une interférence a lieu, elle sera détectée, et les changements engagés seront annulés.

On cherche pour ces structures de données partagées à garantir :

- des propriétés de sûreté (pas de déréférencement de pointeurs non alloué),
- des propriétés fonctionnelles liée à la cohérence temporelle (séquentialisabilité, cohérence des états de repos, linéarisabilité,...)
- des propriétés de progrès, similaires à l'absence de deadlock en concurrence classique. Les algorithmes sont ainsi dits sans attente, sans verrouillage, ou encore sans obstruction.

Notre objet d'étude a consisté en premier lieu à adapter nos méthodes pour la vérification de propriétés de sûreté, les autres propriétés étant pour le moment hors d'étude.

2.2.2 Représentation symbolique des verrous

Nous avons réutilisé la représentation symbolique de la mémoire basée sur les formes mémoires, en l'étendant pour prendre en compte l'état des verrous de chaque cellule d'une liste. L'état d'un verrou représente dans notre cas une information *finie*, puisque le nombre de threads est fixé.

2.2.3 Résultats

Nous sommes en train d'implémenter notre méthode, et nous avons bon espoir de vérifier le programme de *lock-coupling* avec un faible nombre de threads en un temps raisonnable.

Chapitre 3

Autres algorithmes

3.1 Logique de séparation

La logique de séparation [28] est une logique dédiée à l'annotation de programme manipulant des pointeurs, dans la tradition de la preuve de programme à la Hoare-Floyd. Le mot "logique" recouvre dans ce cadre à la fois le langage de spécification et l'ensemble des règles de déduction de triples de Hoare. Dans notre travail, nous nous sommes focalisé sur le *langage de spécification* (même si un travail sur le système de preuve, dans le cadre de programmes concurrents, est en cours de réalisation [6]).

La logique de séparation compte deux connecteurs dits sub-structurels : la conjonction séparante ($*$), et la baguette magique, dont le traitement automatique ne peut être effectué par les démonstrateurs automatiques classiques, qui ne traitent en général que le premier ordre.

Ces connecteurs permettent d'exprimer simplement des propriétés de non aliasing. Par exemple, le triplet $\{x \mapsto 3 * y \mapsto 3\}\mathsf{free}\ x\{y \mapsto 3\}$ exprime que si x et y pointent sans aliasing vers 3 avant libération de x, alors y pointera toujours sur 3 après libération de x.

Notre travail a consisté tout d'abord à étendre cette logique d'états en une logique temporelle, dont nous avons étudié la décidabilité, la complexité et l'expressivité de certains fragments. Les propriétés mémoires exprimables *et* décidables dans cette logique étaient des propriétés de non aliasing, mais les propriétés d'accessibilité par déréférencement (propriétés de listes, par exemples) étaient indécidables. Ces premiers résultats nous ont conduit, dans un second temps, à reconsidérer la logique de séparation en tant que logique d'état, et à étudier sa décidabilité et son expressivité concernant les propriétés d'accessibilité, d'abord sans données, puis avec données. La décidabilité de la logique de séparation (plus précisément du problème de satisfaisabilité) a pour application directe la vérification automatique de préservations d'invariants de boucles, et donc de programmes faiblement annotés.

Nos résultats principaux peuvent se résumer ainsi :

– concernant l'extension temporelle : la décidabilité de la conséquence logique pour les exécutions arbitraires a été établie.
– concernant l'expressivité de la logique de séparation (statique) : la logique de séparation est aussi expressive que son fragment sans conjonction séparante. Ce résultat théorique est très surprenant et non trivial.
– concernant la décidabilité de la logique de séparation : est décidable tant que la baguette magique n'est pas prise en compte. Cette décidabilité s'étend aux propriétés de tri pour des listes avec données. Ce résultat a une application directe dans la preuve automatique de programmes, notamment généralise l'algorithme implémenté dans l'outil SmallFoot [24].

Nous détaillons ci-dessous ces trois axes de recherche ainsi que les résultats obtenus.

3.1.1 Extension temporelle de la logique de séparation

Une approche de la vérification des programmes à pointeurs basée sur la logique de séparation et la logique temporelle a été explorée durant le début de la thèse de Rémi Brochenin, co-encadré par Stéphane Demri et Etienne Lozes [14, 4].

Nous avons cherché à étendre la logique de séparation pour pouvoir spécifier des propriétés sur les exécutions de programmes vues comme des suites d'états ou traces. Une extension naturelle est de rajouter les connecteurs de la logique temporelle linéaire (LTL), qui permettent d'exprimer qu'une certaine propriété ϕ est vérifiée tout le temps ($G\phi$), au bout d'un certain temps ($F\phi$), voire au bout d'un certain temps sachant qu'en attendant une autre propriété ψ est vérifiée ($\psi U \phi$). Nous avons ainsi proposé un formalisme combinant les caractéristiques de la logique de séparation et la logique temporelle, appelé ci-dessous LTL_{mem}.

Nous avons proposé un modèle formel très général qui rend compte à la fois de l'arithmétique des pointeurs (tableaux, matrices,...) et des structures récursives (listes, arbres,...). Nous avons considéré plusieurs problèmes de décision liés à LTL_{mem} : le model-checking, autrement dit savoir si un programme P donné vérifie une spécification ϕ de LTL_{mem}, et la conséquence logique, autrement dit savoir si une spécification en induit automatiquement une autre. Nous avons fait appel à une représentation symbolique de la mémoire comme ensemble maximalement consistant de formules d'états. L'intérêt essentiel de cette abstraction est de rester correcte et complète dans la plupart des cas (elle échoue cependant pour certaines propriétés temporelles sur l'arithmétique des pointeurs), et de fournir une représentation finie de toutes les formes mémoires qui doivent être considérées. Notre technique de vérification du problème de conséquence logique est basée sur une traduction de la logique temporelle vers les automates de Büchi [29] : à toute spécification ϕ on associe un automate A_ϕ capable de reconnaître exactement les

mots, c'est-à-dire les séquences d'états mémoires symboliques, qui satisfont la spécification ϕ. Une fois obtenu l'automate, le problème de la conséquence logique se ramène à celui du test du vide du langage reconnu, qui est connu pour être décidable.

En pratique, nous n'avons pas cherché à implémenter notre algorithme, mais nous avons cherché à connaître sa complexité théorique : bien qu'élevée (PSPACE-complète), elle reste la même que celle de la logique de séparation "statique" et la logique temporelle propositionnelle, ce qui d'un certain point de vue valide l'extension que propose LTL_{mem}.

3.1.2 Expressivité de la logique de séparation

L'étude de l'expressivité d'un formalisme logique ne conduit pas nécessairement à des algorithmes de vérification directs, mais elle peut proposer des traductions vers d'autres problèmes, ou révéler l'indécidabilité d'un problème. Nos résultats d'expressivité ont justement abondé en ce sens [9].

Nous avons considéré la logique de séparation sur les modèles mémoires à un seul sélecteur et sans données, donc un modèle mémoire plus restreint que celui de l'extension temporelle, mais nous avons aussi considéré la quantification du premier ordre. Nous avons d'abord observé que la conjonction séparante couplée à la quantification du premier ordre permettait d'exprimer l'accessibilité, augmentant notablement l'expressivité de la logique. Nous avons ensuite montré que le fragment sans baguette magique admettait une traduction dans la logique monadique du second ordre, connue pour être décidable sur les graphes fonctionnels. En conséquence, la logique de séparation du premier ordre, sans baguette magique, sur les tas mémoires sans données à un sélecteur, est décidable du point de vue du problème de la satisfaisabilité.

Nous avons ensuite étudié la logique de séparation avec baguette magique, qui était connue pour être décidable sans le premier ordre et indécidable au premier ordre sur les modèles avec deux sélecteurs. Nous avons montré que cette logique se traduisait au second ordre (polyadique), et que la logique du second ordre elle-même se traduisait dans la logique de séparation sans conjonction séparante. Ceci nous a permis d'établir que la logique de séparation est aussi expressive que la logique du second ordre, et que la conjonction séparante est redondante et peut s'exprimer à partir de la baguette magique. En particulier, la logique de séparation est indécidable aussi pour les structures à un seul sélecteur.

3.1.3 Décidabilité de la logique de séparation

Le résultat positif de l'étude précédente d'un point de vue vérification est la décidabilité de la logique de séparation sans baguette magique. Ce fragment logique est d'ailleurs celui considéré, avec d'autres restrictions, dans

certains outils basés sur la logique de séparation comme SmallFoot [24]. Nous avons cherché à prolonger ce résultat sur les structures à un sélecteur comportant des données [1]. Nous avons ainsi considéré un modèle mémoire avec données, et des prédicats de comparaison de données permettant d'exprimer notamment l'aspect trié d'une liste avec données. Nous avons alors observé que sans restriction, la logique de séparation (sans baguette magique) est indécidable lorsqu'on ajoute les données. Nous avons alors défini une restriction syntaxique ne permettant de comparer les données que sur des positions mémoires voisines ou fixées à l'avance, ce qui préserve l'expressivité concernant les propriétés de tri, et garantit la décidabilité. L'extension avec baguette magique a été redémontrée indécidable, même en présence de certaines restrictions qui garantissait la décidabilité dans le cas sans données.

Cette logique nous a permis d'exprimer dans le fragment décidable la préservation de l'invariant de boucle d'un programme fusionnant deux listes triées.

3.1.4 Intégration dans L2CA

Une autre approche visant à établir la décidabilité de la logique de séparation, enrichie de prédicats arithmétiques, a été conduite par Radu Iosif et Marius Bozga [8]. L'idée centrale de cette approche consiste à restreindre la quantification du premier ordre à un certain nombre de quantificateurs et une certaine alternance entre existentiels et universels. Cette approche a permis d'établir la décidabilité d'un fragment du premier ordre (sans baguette magique) sur le modèle à un sélecteur sans données, prenant en compte des propriétés arithmétiques sur les longueurs de listes (par exemple, une liste est de longueur la somme de celles de deux autres listes).

Ces résultats ont été ensuite intégrés dans l'outil L2CA, qui admet en entrée une description en logique de séparation de l'état de la mémoire au début du programme.

3.2 Structures de données composites

Nous avons étudié d'autres logiques d'états pour la description de propriété du tas mémoire. Une première approche a consisté à étudier une logique exprimant des contraintes sur les chemins [20] dans le graphe de structures de données à sélecteurs multiples, pouvant contenir des données sur un domaine fini. L'approche que nous présentons ici en détail est la logique CSL [2], qui intègre la plupart des aspects de la mémoire que l'on a cherché à étudier. Dans cette logique, il est possible de parler à la fois :

- de propriétés d'accessibilité pour des graphes mémoires à plusieurs sélecteurs,
- d'arithmétique de pointeurs avec des contraintes sur les index de tableaux décrits dans l'arithmétique de Presburger,

– des contraintes sur les données qui apparaissent dans les tableaux ou les structures récursives, exprimables dans une logique du premier ordre sur le domaine de ces données qui est un paramètre général de cette logique, supposée en pratique décidable.

Pour obtenir la décidabilité de cette logique, nous avons considéré un fragment de la logique dans lequel l'alternance de quantificateurs est restreinte et guidée par le type des cellules mémoires sur lesquelles on quantifie. A chaque type on associe une hauteur dans une certaine hiérarchie, qui n'est pas nécessairement liée à hiérarchie des structures de données récursives (par exemple, dans le cas d'une liste de listes, le type des cellules de la liste de liste peut être soit plus grand, soit plus petit que celui des cellules des listes secondaires). Cette hierarchie de type doit être fixée avec soin, car c'est elle qui définit les restrictions sur les propriétés exprimées : si l'on quantifie en respectant l'ordre cette hiérarchie de type ainsi que d'autres contraintes, on obtient la décidabilité de cette logique. La technique utilisée est une transformation de formules, qui permet d'éliminer les contraintes arithmétiques, puis les contraintes de chemins, pour obtenir une formule du premier ordre sur la logique paramétrique sur les données.

Cette logique a par ailleurs la propriété d'être stable par calcul de plus forte post-condition, et on peut donc automatiser la vérification de préservation d'invariants de boucles exprimables dans cette logique. Nous avons testé notre logique sur des structures de données composites non triviales, telles que des listes à liens multiples, et des *skip lists*.

3.3 Analyse d'accessibilité de programmes multi-thread avec appels de procédures et structures de données dynamiques

L'analyse d'accessibilité avec un changement de contexte borné (ou bounded context switch), est une approche efficace pour la détection d'erreurs dans les programmes multi-thread. En effet, il s'avère que dans beaucoup de cas, les erreurs apparaissent après un nombre assez faible de changements de contexte. Noter qu'il s'agit ici de borner le nombre de changements de contexte d'une thread à une autre, sans borner le nombre d'étapes de calcul de chacune des threads.

Dans [12], nous étudions l'application de cette approche à l'analyse de programmes multi-thread avec :

1. appels de procédures (potentiellement récursives), et
2. manipulation de structures de données dynamiques (création dynamique d'objets et manipulation de pointeurs)

Nous définissons une sémantique des programmes basée sur les automates à pile concurrents, ayant comme symboles de pile ce que nous appelons des

tas mémoire visibles (ou visible heaps). Un tas mémoire visible est la partie du tas mémoire du programme qui est accessible (à un moment donné) à partir des variables globales et des variables locales (de la procédure qui s'exécute à ce moment là).

Nous utilisons des techniques d'analyse d'automates à pile pour définir un algorithme qui explore tout l'espace des configurations accessibles du programme, ceci en fixant :

1. une borne sur le nombre des changements de contexte, et
2. une borne sur la taille des tas mémoire visibles.

Remarquons que, du fait que nous permettons des procédures récursives, d'une part (1) la taille de la pile des appels est non bornée (car, comme il est mentionné plus haut, le nombre des étapes de calcul des threads n'est pas borné entre les changements de contexte), et d'autre part (2) la taille du programme analysé peut ne pas être bornée, même en fixant une borne sur la taille des tas visibles.

Bibliographie

Publications du projet Averiles

— 2009 —

[1] K. Bansal, R. Brochenin, and É. Lozes. Beyond shapes : Lists with ordered data. In L. de Alfaro, editor, *Proceedings of the 12th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'09)*, volume 5504 of *Lecture Notes in Computer Science*, pages 425–439, York, UK, Mar. 2009. Springer.

[2] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In M. Bravetti and G. Zavattaro, editors, *CONCUR 2009 - Concurrency Theory, 20th International Conference, Bologna, Italy, September 1-4, 2009. Proceedings*, volume 5710 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2009.

[3] M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2009.

[4] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. 2009. To appear in a special issue dedicated to selected papers from LFCS'07.

[5] R. Iosif and A. Rogalewicz. Automata-based termination proofs. In S. Maneth, editor, *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, volume 5642 of *Lecture Notes in Computer Science*, pages 165–177. Springer, 2009.

[6] J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In Z. Hu, editor, *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09)*, Lecture Notes in Computer Science, Seoul, Korea, Dec. 2009. Springer. To appear.

— **2008** —

[7] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 341–354. Springer, 2008.

[8] M. Bozga, R. Iosif, and S. Perarnau. Quantitative separation logic and programs with lists. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2008.

[9] R. Brochenin, S. Demri, and É. Lozes. On the almighty wand. In M. Kaminski and S. Martini, editors, *Proceedings of the 16th Annual EACSL Conference on Computer Science Logic (CSL'08)*, volume 5213 of *Lecture Notes in Computer Science*, pages 323–338, Bertinoro, Italy, Sept. 2008. Springer.

[10] P. Habermehl, R. Iosif, and T. Vojnar. A logic of singly indexed arrays. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 558–573. Springer, 2008.

[11] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In R. M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008.Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 474–489. Springer, 2008.

— **2007** —

[12] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV'07*, volume 4590. LNCS, 2007.

[13] M. Bozga and R. Iosif. On flat programs with lists. In *VMCAI'07*, volume 4349 of *LNCS*, pages 122–136. Springer, 2007.

[14] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. In *LFCS'07*, volume 3634 of *LNCS*, pages 100–114. Springer, 2007.

[15] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In K. S. Namjoshi, T. Yoneda,

T. Higashino, and Y. Okamura, editors, *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, volume 4762 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2007.

— 2006 —

[16] S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *AVIS'06*, 2006.

[17] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In Th. Ball and R. B. Jones, editors, *CAV'06)*, volume 4144 of *LNCS*, pages 517–531, Seattle, Washington, USA, Aug. 2006. Springer.

[18] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In K. Yi, editor, *SAS'06)*, volume 4134 of *LNCS*, pages 52–70, Seoul, Korea, Aug. 2006. Springer.

[19] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In H. Hermanns and J. Palsberg, editors, *TACAS'06*, volume 3920 of *LNCS*, pages 350–364, Vienna, Austria, Mar. 2006. Springer.

[20] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In L. Aceto and A. Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006.

Autres publications

[21] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast : Fast acceleration of symbolic transition systems. In *CAV'03*, volume 2725 of *LNCS*, pages 118–121. Springer, 2003.

[22] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In *AVIS'04*, 2004.

[23] S. Bardin, J. Leroux, and G. Point. Fast extended release. In *CAV'06*, volume 4144 of *LNCS*, pages 63–66. Springer, 2006.

[24] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot : Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.

[25] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS'05*, volume 3440 of *LNCS*, pages 13–29, Edinburgh, UK, Apr. 2005. Springer.

[26] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In J. Srba and S. A. Smolka, editors, *INFINITY'05*, volume 149 of *ENTCS*, pages 37–48, San Francisco, CA, USA, Feb. 2006. Elsevier Science Publishers.

[27] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In R. Alur and D. A. Peled, editors, *CAV'04*, volume 3114 of *LNCS*, pages 372–386, Boston, Massachusetts, USA, July 2004. Springer.

[28] J. Reynolds. Separation logic : a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.

[29] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115 :1–37, 1994.