Vérification probabiliste de résultats d'analyse statique

Antoine Bride

Master de recherche en informatique,
ISTIC — ÉNS Cachan, antenne de Bretagne
Rapport

Encadrant : David Cachera
Équipe Celtique
Inria Rennes — Irisa

Vérification probabiliste de résultats d'analyse statique

Antoine Bride

Rennes, 5 juin 2012

Résumé

Ce document traite de la vérification probabiliste de résultats d'analyse statique. Il s'agit plus précisément de mettre au point un algorithme de Monte-Carlo permettant de vérifier qu'un objet (appelé certificat) que l'on nous donne est bien le résultat d'une analyse statique. À notre connaissance, aucun algorithme de la sorte n'existe. Nous avons donc commencé par nous définir un cas précis sur lequel travailler à savoir une analyse *Points-to* sur un langage impératif simple que nous définissons. Nous avons ensuite proposé un algorithme de Monte-Carlo qui soit simple à implémenter et rapide à exécuter pour ce cas en particulier. Une grande partie de notre travail a alors consisté à démontrer que l'algorithme avait de bonne propriétés de convergence en terme de probabilité d'erreur. Nous avons enfin implémenté cet algorithme afin d'obtenir des résultats expérimentaux. Finalement, nous proposons des pistes pour élargir le champ d'application et ne plus se restreindre à un cas précis.

Mots-clefs : Analyse de programmes, Analyse statique, Algorithmique probabiliste, Analyse *Points-to*, Monte-Carlo

Table des matières

1 Introduction

Actuellement, quand on obtient un logiciel, on n'a presque jamais de garantie que le logiciel en question se comporte comme il faut. Le consommateur n'a alors plus qu'une seule alternative : faire confiance au producteur, ou prouver lui-même que le logiciel se comporte bien. Cependant, la plupart des consommateurs n'ont pas la patience ou pas les compétences requises en informatique par faire le second choix. Un moyen de gagner du temps est d'exiger du producteur qu'il réalise une analyse de programme sur son logiciel afin de prouver que ce dernier se comporte bien.

Les méthodes d'analyse de programme sont faites pour étudier le comportement d'un programme, en établissant des propriétés sur celui-ci. On peut séparer ces méthodes en deux grandes catégories. La première catégorie, l'analyse dynamique (ou test), consiste à faire fonctionner un programme afin de voir ce qu'il se passe. Par exemple, on peut lancer un programme avec des valeurs d'entrée arbitraires sur une machine virtuelle et regarder comment la mémoire est affectée. Cette classe d'analyse n'est pas traitée dans ce document car elle n'est pas faite pour garantir que *rien de grave ne peut arriver*.

La seconde classe d'analyse de programmes, l'analyse statique, est faite pour obtenir ce type de garantie. Elle ne nécessite pas l'exécution du programme. Le principe est d'analyser le code[1] d'un programme afin d'établir des propriétés sur son comportement. Un problème important dans ce type d'approche est l'indécidabilité de telles propriétés. Par exemple, la question de savoir si un programme termine ou non est indécidable. En réponse à ce problème, on définit un modèle qui surapproxime le comportement d'un programme. Le but est que les propriétés que l'on cherche à établir soient décidables dans le modèle. Si les propriétés garantissent que le modèle se comporte bien, alors le programme se comporte bien car le modèle le surapproxime. Plus précisément, établir des propriétés sur le modèle consiste souvent à obtenir des équations sur les variables du modèle et à les résoudre. Cette dernière étape est souvent coûteuse en terme de calculs car elle consiste généralement à réaliser un calcul de point fixe sur un treillis.

On peut alors imaginer qu'un producteur fournissant un logiciel `java` réalise une analyse statique (publique) sur le `byte code java` de celui-ci et fournisse donc, en sus du `byte code`, le résultat de l'analyse en tant que certificat. Le producteur garantirait alors le bon comportement de son logiciel sans avoir à en dévoiler le code source. L'acquéreur, lui, pourrait vérifier que le certificat est bien conforme à l'analyse statique faite sur le logiciel.

Partant toujours du principe que l'acquéreur d'un programme n'est pas patient. Il nous semblait important réaliser un algorithme qui puisse s'exécuter rapidement. Pour cette raison, nous avons souhaité réaliser un algorithme probabiliste pour vérifier le certificat. L'inconvénient de tels algorithmes est qu'ils peuvent fournir un résultat erroné (Algorithmes de Monte-Carlo) ou être beaucoup plus longs à terminer que prévu (Algorithmes de Las Vegas). De plus, deux propriétés des algorithmes probabilistes nous ont semblé intéressantes.

La première de ces propriétés est qu'un algorithme probabiliste est souvent, en soit, très simple. Tout la difficulté repose dans la preuve que l'algorithme se comporte bien. Rajeev Motwani et Prabhakar Raghavan expliquent, dans leur ouvrage *Randomized Algorithms* [MR95], que les algorithmes déterministes ont une certaine tendance à toujours être plus compliqués pour réduire leur complexité et à jouer sur des propriétés mathématiques avancées qu'il faut comprendre lors de l'implémentation. Au contraire, les algorithmes probabilistes ont l'avantage de rester simple. Le principe est que dans un programme, il y a souvent un choix à faire qui détermine le succès du programme ou son temps d'exécution. Par exemple, pour l'algorithme de tri rapide, le point dur est le choix du pivot. Les algorithmes probabilistes proposent alors souvent de faire quelque chose de très peu compliqué : choisir au hasard en comptant sur le fait qu'il n'y a que peu de choix vraiment

[1] Code source, code assembleur, `byte code java`...

mauvais. Toute la difficulté est alors de prouver qu'effectivement, choisir au hasard donne de bon résultats. Ceci est transparent pour qui souhaite implémenter l'algorithme.

Le deuxième avantage de cette sorte de paradigme est d'éviter les « mauvaises » entrées. Pour reprendre l'exemple du tri rapide, lorsque l'on connait la manière déterministe dont le pivot est choisi, on peut toujours créer une entrée qui le fera s'exécuter en un temps proche du temps quadratique. Si le pivot est choisit au hasard, en revanche, il est impossible de créer une entrée qui garantira un temps d'exécution quadratique. Le choix aléatoire du pivot peut bien sûr être malchanceux et sélectionner systématiquement le plus grand élément du tableau comme pivot. Cependant, cela ne dépend plus du tableau. Là où l'on compte sur le fait que les entrées sont bien réparties dans le cas déterministe, on ne compte plus que sur le fait que le générateur aléatoire, que l'on contrôle bien plus, donne un bon aléas dans le cas probabiliste.

Plus particulièrement nous nous sommes intéressés à l'élaboration d'un algorithme de Monte-Carlo. Il y a plusieurs raisons pour lesquelles nous avons choisit de faire cela. Premièrement, un tel algorithme permet de choisir à quel point la vérification sera rapide[2]. Deuxièmement, un tel algorithme n'a pas nécessairement besoin de connaître toutes propriétés données par le certificat, seulement celles qu'il vérifie. La quantité de propriétés étant éventuellement grande, cela permettrait de gagner du temps en ne demandant que celles que l'on souhaite vérifier au producteur.

À notre connaissance, aucun algorithme de ce type n'existe à ce jour. La section 2 porte donc sur l'algorithmique probabiliste en général, un algorithme probabiliste dans le domaine de l'analyse de programme et des analyses statiques proches de celle que nous considérons. De plus, nous spécifions formellement le problème que nous nous posons. D'abord nous définissons le langage que nous considérons (section 3), ensuite l'analyse qui est faite sur le langage (section 4). Une fois cela fait, nous pouvons définir notre algorithme (section 5) et discuter de son implémentation et de ses résultats (section 6). Enfin nous parlerons de travaux ultérieurs pouvant être réalisés (section 7) et nous conclurons (section 8).

2 Travaux Connexes

En terme de travaux en ce qui concerne l'algorithmique probabiliste en analyse de programme, on peut citer notamment ceux de Sumit Gulwani et la *Random Interpretation* [Gul05, GN03, GN04a, GN04b, GN05]. Son but est de cumuler les avantages de l'analyse dynamique (ne capturer que le comportement du programme) et de l'analyse statique (capturer l'intégralité du comportement du programme). Pour cela Gulwani propose de faire tourner le programme sur un jeu d'entrées comme en analyse dynamique, mais de s'intéresser à tous les branchements possibles (notamment lors de conditionnelles ou de boucles) pour capturer le maximum de comportements possibles. Afin de toujours garder un seul état des variables à chaque endroit du programme, lorsque deux branches se réunissent, les états courants de chaque branche sont fusionnés grâce à une fonction aléatoire. Il s'agit en réalité de réaliser une somme pondérée des états de poids aléatoire. À savoir, si une variable x à la valeur v_1 dans une branche et v_2 dans un autre, lors de la réunion des deux branches, on tire un poids w aléatoirement et x reçoit la valeur $wv_1 + (1 - w)v_2$. L'intérêt, étant donné que w est tiré une fois pour toutes les variables lors de la fusion de deux branches, est que les relations linéaires[3] sont conservées puisque la somme est elle même linéaire. Gulwani montre dans sa thèse [Gul05] qu'il parvient à obtenir un algorithme pro-

[2] En revanche, plus la vérification est rapide, plus l'algorithme a de chances de se tromper.

[3] C'est à dire les relations du type $a_1x_1 + \cdots + a_nx_n = b$ où les x_i sont des variables du programmes et les a_i ainsi que b sont des constantes.

babiliste qui capture toutes les relations linéaires d'un programme et qui ne capture que celles-là avec une probabilité élevée. Il montre notamment que sa méthode peut s'adapter dans le cas de l'arithmétique affine [GN03], dans l'abstraction de fonctions [GN04a] et de l'utilisation de fonctions déclarées par l'utilisateur [GN05]. Une des principales forces de ses travaux est que l'algorithme qu'il propose est rapide et, en soit, très simple. Seule la preuve que l'algorithme se comporte bien est difficile.

Une remarque que l'on peut faire sur la sous-catégorie des algorithmes probabilistes qui nous intéresse, à savoir les algorithmes de Monte-Carlo, est qu'ils sont souvent utilisés pour répondre à une question fermée. Dans notre cas, est-ce que oui ou non, un certificat correspond à l'analyse d'un programme. Une propriété pratique apparaît si l'on peut montrer que l'algorithme ne se trompe jamais quand il donne l'une de ses réponses. Dans notre cas, imaginons que notre algorithme consiste à vérifier que le certificat est conforme à un sous-ensemble des propriétés données par l'analyse, et réponde oui si c'est le cas, non sinon. Notre algorithme répond non uniquement si il a trouvé au moins une propriété que le certificat ne vérifie pas et ne peut donc pas se tromper dans ce cas. La remarque, faite par Motwani *et al.* [MR95], est alors la suivante : si l'algorithme à une probabilité $p < 1$ de répondre oui à tort, on lance l'algorithme n fois de suite, l'algorithme a alors une probabilité p^n de répondre n fois oui à tort (et si il répond non, une fois, on est fixé sur le résultat). Autrement dit, nous venons d'obtenir presque gratuitement la décroissance exponentielle de la probabilité d'erreur de notre algorithme contre une croissance linéaire de son temps d'exécution. Nous n'avons eu à définir formellement ni l'algorithme, ni les propriétés, ni même l'analyse.

Afin, d'obtenir une quantification précise de cette probabilité d'erreur de l'algorithme, il nous est tout de même nécessaire de définir formellement notre cadre. Nous avons choisi de travailler sur un langage de pointeur, c'est pourquoi nous allons maintenant discuter de quelques manières d'aborder l'analyse *points-to* avant de commencer les définitions formelles en section 3.Une analyse *Points-to* a pour but d'établir vers quelle adresse peut pointer chaque variable. Elle se focalise donc sur toutes les manipulations d'adresse d'un programme.

Parmi d'autres travaux, nous pouvons citer ceux de Rupesh Nasre et Ramaswamy Govindarajan qui proposent de modéliser les équations venant d'une analyse *points-to* par des systèmes d'équations linéaires [NG11]. L'inconvénient est cette méthode est que traduire une équation de l'analyse *points-to* à l'algèbre linéaire a un coût. L'avantage est que la résolution de systèmes d'équation linéaire est un problème ancien auquel beaucoup de solutions efficaces ont été proposées. On peut citer aussi les travaux de John Whaley et Monica S. Lam qui ont proposé d'utiliser des diagrammes de décision binaire pour résoudre les équations d'une analyse *points-to* [WL04]. Ils ont ensuite continué leurs travaux avec l'aide de Dzintars Avots et Michael Carbin pour réaliser un compilateur de `datalog` (un sous-langage de `prolog`) qui est dédié à l'analyse statique en général et à l'analyse *points-to* en particulier [WACL05]. Ce compilateur nous a été utile. En effet, nous avons pu l'utiliser afin d'obtenir de vrais résultats de notre analyse statique. Cela nous a permis de modifier légèrement ces résultats pour tester le comportement de l'algorithme que nous avons implémenté.

3 Un langage de pointeur

Maintenant que d'autres approches d'analyse *points-to* ont été présentées, nous proposons la nôtre. Dans cette section, on définit donc le langage de pointeurs sur lequel une analyse sera faite. On commence par donner la syntaxe du langage (section 3.1), puis sa sémantique (sections 3.2, 3.3 et 3.4).

$\epsilon \in$ **ExprVar**

$\epsilon :=$ $\mid x$ $\quad x \in$ **Var**

$\quad\quad \mid *x$ $\quad x \in$ **Var**

(a) Expressions de variables

$e \in$ **ExprVal**

$e :=$ $\mid x$ $\quad x \in$ **Var**

$\quad\quad \mid *x$ $\quad x \in$ **Var**

$\quad\quad \mid \&x$ $\quad x \in$ **Var**

(b) Expressions de valeurs

$b \in$ **Bool**

$b :=$ $\mid$ true

$\quad\quad \mid$ false

$\quad\quad \mid$ not b $\quad b \in$ **Bool**

$\quad\quad \mid b_1\ o\ b_2$ $\quad b_1, b_2 \in$ **Bool**, $\ o \in \{\wedge, \vee\}$

$\quad\quad \mid \epsilon_1 == \epsilon_2$ $\quad \epsilon_1, \epsilon_2 \in$ **ExprVar**

$\quad\quad \mid e_1 = e_2$ $\quad e_1, e_2 \in$ **ExprVal**

(c) Expressions booléennes

FIG. 1 – Définition des différentes expressions présentes dans le langage

3.1 Syntaxe du langage

La syntaxe du langage de pointeur que l'on souhaite définir est celle d'un langage impératif standard à ceci près qu'elle considère que les variables contiennent des adresses et non des entiers. On définit d'abord les domaines syntaxiques, puis les syntaxes des sous-expressions : expressions de variables, d'adresses, et booléennes. Enfin, on considère la syntaxe d'un programme.

Comme évoqué précédemment, le langage utilisera des variables ; on a donc besoin d'un ensemble de variables **Var**. On définit de plus un ensemble d'étiquettes **Label** qui permettra d'identifier les expressions entre elles.

On considère maintenant les sous-expressions. Pour commencer, on définit l'ensemble des expressions de variables : **ExprVar** (figure 1a). On choisit de pouvoir exprimer une variable soit par la variable elle-même, soit par le déréférencement d'une autre variable à l'aide de l'opérateur $*$.

Ensuite, on définit l'ensemble des expressions de valeurs : **ExprVal** (figure 1b). Une valeur peut être exprimée de trois manières : à l'aide d'une variable, par le truchement d'un déréférencement (opérateur $*$), ou en référençant une variable avec l'opérateur &.

Enfin, on définit l'ensemble des expressions booléennes : **Bool** (figure 1c). Cet ensemble est défini inductivement de manière usuelle à la différence que l'on considère deux égalités. La première est le test d'égalité de variables : $\epsilon_1 == \epsilon_2, \quad \epsilon_1, \epsilon_2 \in$ **ExprVar**. Dans la section 3.3, on définira la sémantique du test == de sorte que si $\epsilon_1 == \epsilon_2$ est évaluée à true, cela signifie que modifier la valeur de ϵ_1 revient à modifier celle de ϵ_2 (et réciproquement). La seconde égalité est le test d'égalité de valeurs : $e_1 = e_2, \quad e_1, e_2 \in$ **ExprVal**.

La syntaxe des sous-expressions ayant été définie, on considère maintenant le langage lui-même. Le langage, **Statement**, possède une syntaxe basique de langage impératif (figure 2). Il contient les instructions suivantes : `skip`, l'assignation d'une valeur à une variable, la séquence, un branchement conditionnel et une boucle. Les instructions sont étiquetées.

La syntaxe du langage ayant été définie, il faut maintenant donner un sens à toutes ces expressions. Dans les sous-sections suivantes, on définit donc la sémantique du programme. On considère en premier lieu les domaines sémantiques 3.2, ensuite les sémantiques des sous-expressions 3.3 et enfin la sémantique du programme lui-même 3.4.

$S \in$ **Statement**, $l \in$ **Label**

$S :=$	$\mid {}^{l}[\,\texttt{skip}]$	
	$\mid {}^{l}[\epsilon := e]$	$\epsilon \in$ **ExprVar** $\quad e \in$ **ExprVal**
	$\mid S_1; S_2$	$S_1, S_2 \in$ **Statement**
	$\mid$ if ${}^{l}[b]$ then S_1 else S_2 endif	$b \in$ **Bool** $\quad S_1, S_2 \in$ **Statement**
	$\mid$ while ${}^{l}[b]$ do S end	$b \in$ **Bool** $\quad S \in$ **Statement**

FIG. 2 – Syntaxe du langage

3.2 Domaines sémantiques du langage

Pour commencer, on définit donc les domaines sémantiques à savoir les ensembles dans lesquels les sémantiques des expressions et du programme prendront leurs arguments et leurs valeurs.

Premièrement, trois ensembles sont définis : **Add**, l'ensemble des adresses utiliséesq **Value** l'ensemble des valeurs que peuvent contenir les zones mémoires (correspondant à des adresses) et {tt,ff} qui correspondent respectivement à vrai et à faux. Étant donné que l'on considère un langage de pointeur, on pose **Value** := **Add**. Cette égalité étant spécifique au cas présent et afin d'éviter toute confusion, nous utilisons tout de même les noms **Add** pour les adresses en tant que telles et **Value** pour les valeurs. Lorsque l'égalité entre ces deux ensembles sera nécessaire, nous l'utiliserons de manière explicite.

Dans le paragraphe précédent nous avons implicitement évoqué l'ensemble des états : **State** = **Add** $\rightarrow$ **Value** $\cup \{\bot\}$. Un état ($\sigma \in$ **State**) associe à chaque adresse une valeur (que contient donc la zone mémoire correspondant à l'adresse) ou $\bot$ si la zone mémoire correspondant à l'adresse n'a pas été initialisée.

Afin d'obtenir une sémantique, il est nécessaire de lier les éléments syntaxiques (**Var**) et sémantiques (**Add** et **Value**). Pour cela, on définit l'ensemble des environnements **Env** = **Var** $\rightarrow$ **Add** $\cup \{\bot\}$ qui à chaque variable, associe l'adresse à laquelle elle correspond (ou $\bot$ si la variable n'a pas d'adresse). Lest environnements sont noté ρ.

Il est désormais possible de définir les sémantiques des sous-expressions.

3.3 Sémantiques des sous-expressions du langage

On définit d'abord la sémantique des expressions de variables, puis celle des expressions de valeurs, et enfin celle des expressions booléennes.

Pour commencer, on s'intéresse donc à la sémantique des expressions de variables, à savoir $\mathcal{E}_a$ (figure 3a). Deux cas peuvent se présenter : x et $*x$ (où $x \in$ **Var**). Lorsque l'on utilise la variable $x \in$ **Var**, on utilise en fait l'adresse qui y correspond : $\rho\, x$. Lorsque l'on déréférence la variable $x \in$ **Var**, on s'intéresse à la valeur que « contient x » à savoir la valeur se trouvant à l'adresse correspondant à x : $\sigma\,(\rho\, x)$. Dans ce dernier cas, il faut que la valeur soit une adresse, c'est le cas car **Value** = **Add**.

Ensuite, on définit la sémantique des expressions de valeurs : $\mathcal{E}_v$ (figure 3b). Trois cas peuvent échoir : x, $*x$, $\&x$ et (où $x \in$ **Var**). La valeur qui correspond à une variable x est celle que contient son adresse, à savoir $\sigma\,(\rho\, x)$. La valeur que l'on souhaite obtenir avec un déréférencement $*x$ est la valeur se trouvant à l'adresse[4] que « contient » x. La sémantique de $*x$ est donc $\sigma\,(\sigma\,(\rho\, x))$ (pour $x \in$ **Var**). Enfin, lorsque l'on référence une variable x avec $\&x$, on s'intéresse simplement à l'adresse de cette variable : $\rho\, x$.

Finalement, on utilise une sémantique des expressions booléennes usuelle, $\mathcal{E}_b$ (figure 3c). Comme annoncé dans la section 3.1, la sémantique de $\epsilon_1 == \epsilon_2$, $\epsilon_1, \epsilon_2 \in$ **ExprVar**,

[4] C'est en réalité une valeur mais **Value** = **Add**.

$$\begin{aligned}\mathcal{E}_a &: \textbf{ExprVar} \to \textbf{State} \to \textbf{Env} \to \textbf{Add}\\ \mathcal{E}_a[x]\,\sigma\,\rho &= \rho\,x\\ \mathcal{E}_a[*x]\,\sigma\,\rho &= \sigma\,(\rho\,x)\end{aligned}$$

(a) Expressions de variables

$$\begin{aligned}\mathcal{E}_v &: \textbf{ExprVal} \to \textbf{State} \to \textbf{Env} \to \textbf{Value}\\ \mathcal{E}_v[x]\,\sigma\,\rho &= \sigma\,(\rho\,x)\\ \mathcal{E}_v[*x]\,\sigma\,\rho &= \sigma\,(\sigma\,(\rho\,x))\\ \mathcal{E}_v[\&x]\,\sigma\,\rho &= \rho\,x\end{aligned}$$

(b) Expressions de valeurs

$$\begin{aligned}\mathcal{E}_b &: \textbf{Bool} \to \textbf{State} \to \textbf{Env} \to \{\text{tt}, \text{ff}\}\\ \mathcal{E}_b[\text{true}]\,\sigma\,\rho &= \text{tt}\\ \mathcal{E}_b[\text{false}]\,\sigma\,\rho &= \text{ff}\\ \mathcal{E}_b[\epsilon_1 == \epsilon_2]\,\sigma\,\rho &= \begin{cases}\text{tt} & \text{si } (\mathcal{E}_a[\epsilon_1]\,\sigma\,\rho) = (\mathcal{E}_a[\epsilon_2]\,\sigma\,\rho)\\ \text{ff} & \text{sinon}\end{cases}\\ \mathcal{E}_b[e_1 = e_2]\,\sigma\,\rho &= \begin{cases}\text{tt} & \text{si } (\mathcal{E}_v[e_1]\,\sigma\,\rho) = (\mathcal{E}_v[e_2]\,\sigma\,\rho)\\ \text{ff} & \text{sinon}\end{cases}\\ \mathcal{E}_b[b_1\,o\,b_2]\,\sigma\,\rho &= (\mathcal{E}_b[b_1]\,\sigma\,\rho)\,[\![o]\!]\,(\mathcal{E}_b[b_2]\,\sigma\,\rho)\end{aligned}$$

(c) Expressions booléennes

FIG. 3 – Sémantique des différentes expressions présentes dans le langage

est évaluée à tt lorsque $\mathcal{E}_a[\epsilon_1]\,\sigma\,\rho = \mathcal{E}_a[\epsilon_2]\,\sigma\,\rho$ et à ff sinon. Donc, si on met une nouvelle valeur dans l'adresse correspondant à ϵ_1, on change la valeur dans l'adresse correspondant à ϵ_2 puisqu'il s'agit de la même.

Les sémantiques des sous-expressions étant définies, on peut désormais s'intéresser à la sémantique d'un programme $S \in \textbf{Statement}$.

3.4 Sémantique du Langage

Notre but est de vérifier qu'un programme n'atteint jamais un état « dangereux ». C'est pour cela que l'on s'intéresse à la sémantique collectrice. Il s'agit en fait de collecter tous les états possiblement atteignables lors de l'exécution d'un programme.

Pour réaliser une telle sémantique, on procède en deux étapes : d'abord on construit un graphe de flot de contrôle à partir de S, puis on « navigue » sur un tel graphe en respectant les conditions données par les arrêtes pour obtenir la sémantique du programme. La raison d'un tel choix est que cela permet d'obtenir une séparation claire entre les éléments structurant d'un programme (tel que la boucle) et les éléments affectant la mémoire. En effet, les premiers sont capturés dans la structure du graphe tandis que les seconds sont gérés par la façon dont le graphe est parcouru.

On s'intéresse à la première étape. On souhaite donc générer un graphe de flot de contrôle `cfg` S que l'on définit par un quadruplet ($\textbf{Vertex}, e, f, \textbf{Edge}$) où **Vertex** est l'ensemble des nœuds du graphe, $e, f \in \textbf{Vertex}$ sont respectivement l'entrée et la sortie du graphe, et **Edge** l'ensemble des arêtes du graphe. On pose que les nœuds sont les étiquettes du programme, donc $\textbf{Vertex} := \textbf{Label}$. Deux instructions qui partagent la même étiquette ne sont donc pas distinguées. En ce qui concerne les arêtes, on définit d'abord $\textbf{Inst} = \{\text{skip}, \epsilon := e, \text{assert}\,b\}$. Cet ensemble comprend les « instructions » atomiques d'un programme, celles ayant un effet ou des conditions sur l'état courant. Le principe est de lier les arêtes à des instruction de sorte que l'on ne prendra les arêtes qu'en respectant l'effet des instructions sur la mémoire.

Les nœuds ont été définis, e est simplement la première étiquette rencontrée dans S (appelée entry S), f est choisie arbitrairement ; seules les arêtes sont toujours à définir. On

cfg_l : **Statement** $\to \mathcal{P}$(**Label** $\times$ **Inst** $\times$ **Label**)

$$
\begin{array}{lcl}
\texttt{cfg}_{l'}\ (^{l}[\epsilon := e]) & = & \{(l, \epsilon := e, l')\} \\
\texttt{cfg}_{l'}\ (^{l}[\text{skip}]) & = & \{(l, \text{skip}, l')\} \\
\texttt{cfg}_{l'}\ (\text{if } ^{l}[b] \text{ then } S_1 \text{ else } S_2 \text{ endif}) & = & \{(l, \text{assert } b, \text{entry } S_1)\} \cup \texttt{cfg}_{l'}\ S_1 \\
 & & \cup \{(l, \text{assert (not } b), \text{entry } S_2)\} \cup \texttt{cfg}_{l'}\ S_2 \\
\texttt{cfg}_{l'}\ (\text{while } ^{l}[b] \text{ do } S \text{ end}) & = & \{(l, \text{assert } b, \text{entry } S)\} \cup \texttt{cfg}_{l}\ S \\
 & & \cup \{(l, \text{assert (not } b), l')\} \\
\texttt{cfg}_{l'}\ (S_1; S_2) & = & \texttt{cfg}_{\text{entry}S_2}\ S_1 \cup \texttt{cfg}_{l'}\ S_2
\end{array}
$$

FIG. 4 – Génération des arrêtes du graphe de flot de contrôle d'un programme

définit les arêtes comme des triplets de **Vertex** $\times$ **Inst** $\times$ **Vertex**. Intuitivement, on souhaite que l'arête (v_1, i, v_2) corresponde au fait de pouvoir aller de l'étiquette v_1 à l'étiquette v_2 en effectuant l'instruction i. On génère les arêtes de $\texttt{cfg}S$ inductivement en fonction des cas pouvant échoir à l'aide de fonction cfg_l (figure 4) où l en indice est une étiquette. Le principe est de garder en mémoire où l'on souhaite se rendre (avec l'indice) et de construire les arêtes permettant de se rendre de l'étiquette courante (donnée par le programme en argument) vers l'étiquette cible (indice). Par exemple $\texttt{cfg}_{l'}$ (while $^{l}[b]$ do S end) est la réunion de trois ensembles :

- $\{(l, \text{assert } b, \text{entry } S)\}$ car si b est vérifié on se branche sur le corps de la boucle : S,
- $\texttt{cfg}_l\ S$ car lorqu'on exécute le corps de la boucle et on revient au test,
- $\{(l, \text{assert (not } b), l')\}$ car si b n'est pas vérifié (assert (not b)) on sort de la boucle.

Au final le graphe de flot de contrôle d'un programme $S \in$ **Statement** est le suivant : $\texttt{cfg}\ S = (\textbf{Label}, \text{entry}\ S, f, \texttt{cfg}_f\ S)$.

Il faut maintenant donner une sémantique à ce graphe de flot de contrôle. Pour cela, on établit des règles quant à la manière de parcourir le graphe à l'aide d'une fonction correspondant à chaque instruction que peut contenir une arête :

$$\xrightarrow{i}: (\textbf{State} \times \textbf{Env}) \to (\textbf{State} \times \textbf{Env}) \qquad i \in \textbf{Inst}$$

$$\frac{}{(\sigma, \rho) \xrightarrow{\text{skip}} (\sigma, \rho)} \qquad \frac{\mathcal{E}_b[b] = \text{tt}\,\sigma\,\rho}{(\sigma, \rho) \xrightarrow{\text{assert } b} (\sigma, \rho)} \qquad \frac{a = \mathcal{E}_a[\epsilon]\,\sigma\,\rho \quad v = \mathcal{E}_v[e]\,\sigma\,\rho}{(\sigma, \rho) \xrightarrow{\epsilon := e} (\sigma[x \mapsto a], \rho)}$$

Le sens que l'on donne est qu'une arrête avec l'instruction skip ne change rien, qu'une arrête ayant l'instruction assert b ne change rien mais ne peut être empruntée que si b est vérifiée, et qu'une arrête avec l'instruction $\epsilon := e$ modifie l'état selon les évaluations de ϵ et e lorsqu'elle est empruntée.

On parcoure alors le graphe ainsi : Si l'on est dans l'état σ_1 et l'environnement ρ_1 au nœud l_1, on peut arriver dans l'état σ_2 et l'environnement ρ_2 au nœud l_2 si et seulement si l'arrête (l_1, i, l_2) existe et l'on a $(\sigma_1, \rho_1) \xrightarrow{\text{i}} (\sigma_2, \rho_2)$. Cela est défini formellement par la fonction suivante :

$$\to_{\text{cfg}}: (\textbf{Label} \times \textbf{State} \times \textbf{Env}) \to (\textbf{Label} \times \textbf{State} \times \textbf{Env})$$

$$\frac{(k_1, i, k_2) \in \mathrm{cfg}_f\ S \quad \sigma_1 \xrightarrow{i} \sigma_2}{(k_1, \sigma_1, \rho) \to_{\text{cfg}} (k_2, \sigma_2, \rho)}$$

Maintenant que l'on sait quels états peuvent être atteints, on peut définir la sémantique collectrice en chaque étiquette d'un programme P :

$$\forall k \in \textbf{Label}, [\![P]\!]^{\text{col}}_k = \{\, \sigma \mid \exists (l, \sigma_{(0,l)}, \rho) \to^*_{\text{cfg}} (k, \sigma, \rho)\}$$

Où $\sigma_{(0,k)}$ est l'état initial associé à chaque étiquette et est une donnée du problème. On peut par exemple n'avoir rien d'initialisé au début : $\forall l \in \textbf{label}, \forall a \in \textbf{Add}, \ \sigma_{(0,l)}\, a = \bot$. On note $[\![P]\!]^{\text{col}} \in \mathcal{P}(\textbf{State})^{\#\textbf{Label}}$ la sémantique collectrice (globale) dont la $k^{\text{ième}}$ composante est la sémantique collectrice de l'étiquette k : $[\![P]\!]^{\text{col}}_k$. On remarque que $(\mathcal{P}(\textbf{State})^{\#\textbf{Label}}, \cup, \cap, \subseteq)$ est un treillis fini.

Il devient alors intéressant d'écrire $[\![P]\!]^{\text{col}}$ comme solution du système d'équation suivant :

$$\forall k \in \textbf{Label}, X_k = X_k^{\text{init}} \cup \bigcup_{(k',i,k)\in P} [\![i]\!] X_{k'} \qquad [\![i]\!] : X \mapsto \{\, \sigma_2 \mid \exists \sigma_1 \in X, \sigma_1 \xrightarrow{i} \sigma_2 \} \tag{1}$$

Où X_k^{init} correspond à $\sigma_{(0,k)}$. En effet, le système d'équations ci-dessus peut être vu comme une fonction croissante dans le treillis auquel appartient $[\![P]\!]^{\text{col}}$. Le fait que le treillis soit fini implique qu'en itérant successivement la fonction sur son résultat (en partant des états initiaux), on atteindra son plus petit point fixe en un nombre d'étapes fini. Autrement dit, on dispose d'une manière de résoudre le système d'équation 1.

4 Sémantique abstraite du langage

Étant donné qu'il est difficile d'exprimer simplement les propriétés dans la sémantique collectrice, nous souhaitons nous en abstraire. Pour cela, nous utilisons une sémantique abstraite. Il s'agit d'approximer la sémantique réelle du programme de façon à travailler dans des ensembles (et des treillis) plus simples. L'inconvénient est que l'on perd en précision. L'intérêt est que l'on gagne en complexité. Nous allons donc définir une telle sémantique abstraite puis montrer qu'elle a de bonne propriétés par rapport à la sémantique collectrice. Ensuite, nous allons analyser le système d'équations abstrait pour définir précisément à quelle question devra répondre l'algorithme que nous souhaitons réaliser.

4.1 Abstraction par rapport à la sémantique de langage

Afin de réaliser la sémantique abstraite, on commence par définir ses éléments constitutifs : les états abstraits notés φ. Ces états abstraits associent aux adresses l'ensemble des valeurs qu'elles peuvent contenir. Ainsi, pour chaque étiquette, on pourra considérer un seul état abstrait plutôt qu'un ensemble d'états :

$$\begin{array}{rccc} \varphi : & \textbf{Add} & \to & \mathcal{P}(\textbf{Value}) \\ & a & \mapsto & \bigcup\limits_{\sigma\in X} \{\sigma\, a\} \end{array}$$

La sémantique abstraite que nous souhaitons définir est alors la sémantique suivante : $[\![P]\!]^{\#} \in (\textbf{Add} \to \mathcal{P}(\textbf{Value}))^{\#\textbf{Label}}$ dont la $k^{\text{ième}}$ composante serait l'état abstrait de l'étiquette k. Elle sera plus simple que la sémantique collectrice puisqu'elle ne comportera qu'un état abstrait dans chaque composante et non un ensemble d'états. On diminue donc le taux d'indirection afin d'obtenir un élément atomique de la sémantique (à savoir un état pour la sémantique collectrice et un état abstrait pour la sémantique abstraite).

Afin de pouvoir atteindre ce but, nous allons munir l'ensemble des sémantiques abstraites $(\textbf{Add} \to \mathcal{P}(\textbf{Value}))^{\#\textbf{Label}}$ d'une structure de treillis : $((\textbf{Add} \to \mathcal{P}(\textbf{Value}))^{\#\textbf{Label}}, \sqcup, \sqcap, \sqsubseteq)$. L'objecif est d'obtenir, au final, une insertion de Galois entre $([\![P]\!]^{\text{col}}, \cup, \cap, \subseteq)$ et $([\![P]\!]^{\#}, \sqcup, \sqcap, \sqsubseteq)$. En effet, cela apportera comme propriété que la solution au système d'équation 1 dans $(\textbf{Add} \to \mathcal{P}(\textbf{Value}))^{\#\textbf{Label}}$ est une surapproximation de la solution dans $[\![P]\!]^{\text{col}}$. Donc, si la solution $[\![P]\!]^{\#}$ n'atteint pas d'état interdit, la solution $[\![P]\!]^{\text{col}}$ non plus.

Pour doter $(\textbf{Add} \to \mathcal{P}(\textbf{Value}))^{\#\textbf{Label}}$ d'une structure de treillis, on définit d'abord la borne supérieure ainsi : $\varphi \sqcup \varphi' = (a \mapsto (\varphi\, a) \cup (\varphi' a))$. Autrement dit, la borne supérieure de

deux états abstraits φ_1 et φ_2 est l'état abstrait qui à une adresse associe toutes les valeurs qui sont associées à φ_1 ou à φ_2 et seulement celles-là. On définit ensuite la borne inférieure comme ceci : $\varphi \sqcap \varphi' = (a \mapsto (\varphi\, a) \cap (\varphi' a))$. Ce qui signifie que la borne inférieure de deux états abstraits φ_1 et φ_2 est l'état abstrait qui a une adresse associe toutes les valeurs qui sont associées à φ_1 et à φ_2 et uniquement celles-là. Enfin, on utilise la relation d'ordre suivante : $\varphi \sqsubseteq \varphi'$ si et seulement si $(\forall a, (\varphi\, a) \subseteq (\varphi' a))$. Pour $\sigma \in \mathbf{State}$, on notera $\sigma \sqsubseteq \varphi$ lorsque $(a \mapsto \{\sigma\, a\}) \sqsubseteq \varphi$ par abus de notation.

Pour montrer qu'il s'agit bien d'une insertion de Galois on définit explicitement la fonction $\alpha : [\![P]\!]^{\text{col}} \to [\![P]\!]^{\#}$ et l'injection $\gamma : [\![P]\!]^{\#} \hookrightarrow [\![P]\!]^{\text{col}}$. On pose $\alpha : X_i \mapsto (a \mapsto \{\sigma\, a \,|\, \sigma \in X_i\})$ noté φ_i. On voit ici que l'on agglomère les états, en perdant un peu de précision. En effet, si l'on considère $\mathbf{Add} = \{1, 2\}$, on ne distingue plus $X = \{\sigma_1 : (1 \mapsto 1,\ 2 \mapsto 2),\ \sigma_2 : (1 \mapsto 2,\ 2 \mapsto 1)\}$ de $Y = \{\sigma_3 : (1 \mapsto 1,\ 2 \mapsto 1),\ \sigma_4 : (1 \mapsto 2,\ 2 \mapsto 2)\}$ car $\alpha\, X = \alpha\, Y = \varphi : (1 \mapsto \{1, 2\},\ 2 \mapsto \{1, 2\})$. On pose, ensuite, $\gamma : \varphi_i \mapsto \{\sigma \,|\, \forall a \in \mathbf{Add},\ (\sigma\, a) \in (\varphi_i\, a)\}$. Il s'agit de considérer tout les états qui peuvent contribuer à obtenir l'état abstrait via α. Pour continuer l'exemple précédent : $\gamma\, \varphi = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$. On remarque que γ est bien injective puisque bijective sur son image d'application réciproque α.

Il faut maintenant montrer que ces fonctions définissent bien une insertion de Galois, donc que $(\alpha\, X_i) \sqsubseteq \varphi_j \Leftrightarrow X_i \subseteq (\gamma\, \varphi_j)$. Pour cela, nous montrons la double implication :

- $(\alpha\, X_i) \sqsubseteq \varphi_j \Rightarrow X_i \subseteq (\gamma\, \varphi_j)$

$$\begin{array}{rcl} (\alpha\, X_i) \sqsubseteq \varphi_j & \Rightarrow & \forall a \in \mathbf{Add},\ (\alpha\, X_i\, a) \subseteq (\varphi_j\, a) \\ & \Rightarrow & \forall \sigma,\ (\forall a \in \mathbf{Add},\ (\sigma\, a) \in (\alpha\, X_i\, a)) \\ & & \qquad \Rightarrow (\forall a \in \mathbf{Add},\ (\sigma\, a) \in (\varphi_j\, a)) \\ & \Rightarrow & \gamma\, (\alpha\, X_i) \subseteq \gamma\, \varphi_j \end{array}$$

Or, par définition, $\forall \sigma \in X_i,\ \forall a \in \mathbf{Add},\ (\sigma\, a) \in \alpha\, X_i\, a$
donc $X_i \subseteq \gamma\, (\alpha\, X_i) \subseteq \gamma\, \varphi_j$

□

- $X_i \subseteq (\gamma\, \varphi_j) \Rightarrow (\alpha\, X_i) \sqsubseteq \varphi_j$

$$\begin{array}{rcl} X_i \subseteq (\gamma\, \varphi_j) & \Rightarrow & (\alpha X_i) \sqsubseteq \alpha(\gamma\, \varphi_j) = \varphi_j \end{array}$$

□

Ceci montre que l'on a un insertion de Galois entre $([\![P]\!]^{\text{col}}, \cup, \cap, \subseteq)$ et $([\![P]\!]^{\#}, \sqcup, \sqcap, \sqsubseteq)$. Il est donc pertinent de calculer $[\![P]\!]^{\#}$ dans la mesure où l'on sait que si elle ne possède pas d'état abstrait dangereux alors $[\![P]\!]^{\text{col}}$ ne possède pas d'état dangereux non plus.

4.2 Analyse dans la sémantique abstraite

Afin de calculer $[\![P]\!]^{\#}$, on calcule de l'image du système d'équation 1 par α. Pour cela, on utilise le fait que $\alpha(\mathbf{A} \cup \mathbf{B}) = (\alpha\mathbf{A}) \sqcup (\alpha\mathbf{B})$. o, obtient :

$$\forall k \in \mathbf{Label}, \varphi_k = \varphi_k^{\text{init}} \sqcup \bigsqcup_{(k',i,k) \in P} [\![i]\!]^{\#} \varphi_{k'} \qquad \text{où } [\![i]\!]^{\#} : \varphi \mapsto (a \mapsto \bigcup_{\substack{\sigma_1 \sqsubseteq \varphi \\ \sigma_1 \xrightarrow{i} \sigma_2}} (\sigma_2\, a))$$

De plus, comme on ne peut plus calculer qu'une surapproximation de la sémantique collectrice, seul un sens de l'égalité nous intéresse, à savoir que le membre de droite est supérieur au membre de gauche. En effet, si une sémantique vérifie une telle inégalité, cela signifie que l'on a une surapproximation de la sémantique abstraite et donc de la sémantique collectrice. Cela implique que de la même manière que si la sémantique abstraite n'atteint pas d'état dangereux alors la sémantique collectrice non plus, si une sémantique vérifiant

une telle inégalité n'atteint aucun état dangereux, c'est aussi le cas pour la sémantique collectrice. Au final on cherche à résoudre :

$$\forall k \in \mathbf{Label}, \varphi_k \sqsupseteq \varphi_k^{\text{init}} \sqcup \bigsqcup_{(k',i,k)\in P} [\![i]\!]^\# \varphi_{k'} \qquad \text{où } [\![i]\!]^\# : \varphi \mapsto (a \mapsto \bigcup_{\substack{\sigma_1 \sqsubseteq \varphi \\ \sigma_1 \xrightarrow{i} \sigma_2}} (\sigma_2\, a)) \qquad (2)$$

De la même manière que pour la sémantique collectrice, le système d'équations peut être vu comme une fonction croissante dans un treillis fini. On peut donc calculer une solution en itérant la fonction (correspondant au système) jusqu'à atteindre un point fixe.

On détail maintenant à quoi correspond le système d'équations en observant ce que les fonctions $([\![i]\!]^\#\varphi_{k'}) \sqsubseteq \varphi_k$ signifient en fonction de l'instruction i :

$$\begin{array}{ll}
i = \text{skip} & \varphi_{k'} \sqsubseteq \varphi_k \\
i = \text{assert}\, b & \forall \sigma \sqsubseteq \varphi_{k'},\, \mathcal{E}_b[b]\,\sigma\,\rho \Rightarrow \sigma \sqsubseteq \varphi_k \\
i = \epsilon := e & \forall \sigma \sqsubseteq \varphi_{k'}, \left\{ \begin{array}{lcl} \rho\,(\mathcal{E}_v[\epsilon]\,\sigma\,\rho) & \mapsto & \{\mathcal{E}_a[e]\,\sigma\,\rho\} \\ a \neq (\mathcal{E}_v[\epsilon]\,\sigma\,\rho) & \mapsto & \varphi_{k'}\, a \end{array} \right. \sqsubseteq \varphi_k
\end{array}$$

Afin de simplifier ces équations nous avons choisi de faire deux hypothèses : on part du principe que rien n'est initialisé au début et on prend le cas d'une analyse insensible au flot. La première hypothèse signifie que $\forall k$, $\forall a \in \mathbf{Add}$, $\varphi_k^{\text{init}}\, a = \bot$, autrement dit φ_k^{init} est élément neutre pour la borne supérieure dans le treillis, donc peut être enlevé des équations. La seconde hypothèse signifie que **Label** est réduit à un seul élément : **Label** = {.}. Les équations deviennent :

$$\varphi_. \sqsupseteq \varphi_. \cup \bigcup_{(.,i,.)\in P} [\![i]\!]^\# \varphi_.$$

Des cas selon i sont devenus triviaux. En effet, pour i = skip, l'équation signifie maintenant $\varphi_. \sqsubseteq \varphi_.$ ce qui est une tautologie. De plus, pour i = assert b , l'équation devient $\forall \sigma \sqsubseteq \varphi_.,\, \mathcal{E}_b[b]\,\sigma\,\rho \Rightarrow \sigma \sqsubseteq \varphi_.$. Autrement dit, des éléments plus petits que $\varphi_.$ sont plus petits que $\varphi_.$. Ceci est à nouveau une tautologie.

Le seul cas restant est $i = \epsilon := e$. On s'intéresse aux cas selon ϵ et e ($x, y \in \mathbf{Var}$) :

$x := y$	$\varphi_.\,(\rho\, y) \subseteq \varphi_.\,(\rho\, x)$
$x := *y$	$\forall v \in \mathbf{Add},\, v \in \varphi_.\,(\rho\, y) \Rightarrow \varphi_.\, v \subseteq \varphi_.\,(\rho\, x)$
$x := \&y$	$\rho\, y \in \varphi_.\,(\rho\, x)$
$*x := y$	$\forall v \in \mathbf{Add},\, v \in \varphi_.\,(\rho\, x) \Rightarrow \varphi_.\,(\rho\, y) \subseteq \varphi_.\, v$
$*x := \&y$	$\forall v \in \mathbf{Add},\, v \in \varphi_.\,(\rho\, x) \Rightarrow \rho\, y \in \varphi_.\, v$
$*x := *y$	$\forall v_1, v_2 \in \mathbf{Add},\, (v_1 \in \varphi_.\,(\rho\, x) \wedge v_2 \in \varphi_.\,(\rho\, y)) \Rightarrow \varphi_.\, v_2 \subseteq \varphi_.\, v_1$

On remarque que tous les quantificateurs universels s'appliquent à des ensembles finis (*e.g.* du type $\varphi_.\,(\rho\, x)$ où $x \in \mathbf{Var}$). Notons $a_1, \ldots, a_A$ les éléments de **Add** (où $A = \#\mathbf{Add}$ le nombre d'éléments de **Add**). Notons $v_{i,1}, \ldots, v_{i,V_i}$ les éléments de $\varphi_.\, a_i$ (où $V_i = \#(\varphi_.\, a_i)$). On peut voir les équations obtenues dans la deuxième colonne de la table 1. Enfin, on peut se ramener à un test d'inclusion pour chaque cas en réalisant des unions et intersections idoines comme le montre la troisième colonne de la table 1. Ce sont ces équations que nous souhaitons vérifier à l'aide d'un algorithme probabiliste.

5 Algorithme de vérification de résultats de l'analyse

Nous pouvons définir la problématique précisément : on est capable de générer les équations correspondant à un programme (table 1), l'environnement ρ est connu, on nous

TAB. 1 – simplifications successives des équations à vérifier.

$x := y$	$\varphi_\cdot(\rho\, y) \subseteq \varphi_\cdot(\rho\, x)$	$\varphi_\cdot(\rho\, y) \subseteq \varphi_\cdot(\rho\, x)$
$x := *y$	Soit $a_i = \rho\, y$: $\varphi_\cdot\, v_{i,1} \subseteq \varphi_\cdot(\rho\, x)$ $\vdots$ $\varphi_\cdot\, v_{i,V_i} \subseteq \varphi_\cdot(\rho\, x)$	Soit $a_i = \rho\, y$: $\bigcup_{k=1}^{V_i} \varphi_\cdot\, v_{i,k} \subseteq \varphi_\cdot(\rho\, x)$
$x := \&y$	$\rho\, y \in \varphi_\cdot(\rho\, x)$	$\rho\, y \in \varphi_\cdot(\rho\, x)$
$*x := y$	Soit $a_i = \rho\, x$: $\varphi_\cdot(\rho\, y) \subseteq \varphi_\cdot\, v_{i,1}$ $\vdots$ $\varphi_\cdot(\rho\, y) \subseteq \varphi_\cdot\, v_{i,V_i}$	Soit $a_i = \rho\, x$: $\varphi_\cdot(\rho\, y) \subseteq \bigcap_{k=1}^{V_i} \varphi_\cdot\, v_{i,k}$
$*x := \&y$	Soit $a_i = \rho\, x$: $\rho\, y \subseteq \varphi_\cdot\, v_{i,1}$ $\vdots$ $\rho\, y \subseteq \varphi_\cdot\, v_{i,V_i}$	Soit $a_i = \rho\, x$: $\rho\, y \subseteq \bigcap_{k=1}^{V_i} \varphi_\cdot\, v_{i,k}$
$*x := *y$	Soient $a_i = \rho\, x$, $a_j = \rho\, y$: $\varphi_\cdot\, v_{j,1} \subseteq \varphi_\cdot\, v_{i,1} \cdots \varphi_\cdot\, v_{j,V_j} \subseteq \varphi_\cdot\, v_{i,1}$ $\vdots \quad \ddots \quad \vdots$ $\varphi_\cdot\, v_{j,1} \subseteq \varphi_\cdot\, v_{i,V_i} \cdots \varphi_\cdot\, v_{j,V_j} \subseteq \varphi_\cdot\, v_{i,V_i}$	Soient $a_i = \rho\, x$, $a_j = \rho\, y$: $\bigcup_{k=1}^{V_j} \varphi_\cdot\, v_{j,k} \subseteq \bigcap_{k=1}^{V_i} \varphi_\cdot\, v_{i,k}$

fournit un état abstrait $\varphi_\cdot$ (le certificat), et on souhaite vérifier que l'état abstrait vérifie les équations. Nous concevons pour le résoudre un algorithme de Monte-Carlo. Pour cela, nous commençons par discuter des choix de conception qui peuvent être faits (section 5.1). Nous calculons ensuite une valeur théorique à la probabilité d'erreur de l'algorithme (sections 5.2 et 5.3). Enfin nous montrons qu'une hypothèse faite lors de du calcul de la valeur théorique n'est pas nécessaire (section 5.4).

5.1 Conception de l'algorithme

Nous souhaitons réaliser un algorithme probabiliste. Il ne faut donc pas perdre de vue que l'on cherche à réaliser un algorithme simple et efficace à implémenter. Étant données, les équations de la table 1, on voit que l'on peut réaliser au moins trois actions de façon probabiliste. Premièrement, on peut vérifier partiellement si une inclusion donnée est vérifiée, ce sera traité section 5.2. Deuxièmement, on peut ne vérifier qu'une partie des inclusions, on s'intéresse à cela en section 5.3. Troisièmement, on peut calculer approximativement les réunions et intersections, nous ne traiterons pas de cela ici. Enfin, afin de garder un algorithme le plus simple possible et de limiter l'espace nécessaire, nous décidons de ne pas garder en mémoire ce qui a déjà été vérifié. L'algorithme 1 que nous obtenons a deux paramètres le nombre d'inclusions testées N et le nombre d'appartenances testées par inclusion L.

Si l'algorithme trouve une appartenance fausse, il s'arrête et rejette le certificat. S'il termine sans trouver d'appartenance fausse, il accepte le certificat. On remarque que d'après nos choix de conception, l'algorithme peut tester plusieurs fois la même inclusion ou la même appartenance. On remarque de plus que notre algorithme de Monte-Carlo ne peut pas se tromper quand il annonce qu'un certificat est faux. En effet, cela signifie que l'algo-

Input : Ensemble d'équations (du type table 1), certificat à vérifier.
Output : Oui si le certificat vérifie le équations, non sinon.

```
Réaliser les unions et intersections dans les équations (cf. table 1)
for n ← 1 to N do
    Choisir aléatoirement une inclusion à vérifier A_n ⊆ B_n
    for l ← 1 to N do
        Choisir aléatoirement un élément a_l ∈ A_n
        if a_l ∉ B_n then
            terminer l'alogrithme, répondre non
terminer l'algorithme, répondre oui
```

Algorithme 1: Notre algorithme.

rithme a trouvé une inclusion fausse.

Nous nous intéressons à quatre propriétés sur notre algorithme : sa probabilité d'erreur, le nombre de bit aléatoires qu'il génère (car cela a un coût), sa complexité spatiale et sa complexité temporelle. La probabilité d'erreur théorique va occuper toute la suite de cette section (5.2, 5.3, 5.4) dans laquelle on obtient une formule théorique dans le cas où les ensembles sont générés aléatoirement. Le nombre de bit aléatoires générés, la complexité spatiable et la complexité temporelles dépendent de l'implémentation, ils seront donc évalués dans la section 6. On note cependant qu'un algorithme testant N inclusions et L appartenances par inclusion réalise en tout $N \times L$ tests d'appartenance. L'évaluation de la complexité temporelle consiste donc principalement à estimer le temps nécessaire à la réalisation d'un test d'appartenance.

À ce propos, on remarque que par rapport à un algorithme testant N inclusions et L appartenances par inclusion, l'algorithme testant $N \times L$ inclusions et 1 appartenance par inclusion réalise autant de tests d'appartenance mais fait plus jouer l'aléa car il est moins restrictif sur la répartition des appartenances testées. Intuitivement, on peut donc penser qu'il est plus intéressant de maximiser le premier paramètre et de garder le second à 1. Cette intuition sera confirmée par les résultats théoriques et pratiques.

Puisque l'on va calculer des probabilités, on définit ici l'espace des événements Ω. On rappelle que les ensembles **A** et **B** sont des sous-ensembles de **Value**. Soient , $v_1, \ldots, v_{\#\mathbf{Value}}$ les éléments de **Value**, on pose $\Omega_1 = \{v_k \in \mathbf{E} \,|\, k \in [\![1, \#\mathbf{Value}]\!] \text{ et } \mathbf{E} \in \mathcal{P}(\mathbf{Value})\}$. De plus, soient $e_1, \ldots, e_{\#\mathbf{E}}$ les éléments d'un ensemble **E** ; on pose $\Omega_{\mathbf{E}} = \{$"on choisit e_k au l-ième tirage" $| \, k \in [\![1, \#\mathbf{E}]\!]$ et $l \in [\![1, L]\!]\}$. Enfin, soient $I_1, \ldots, I_{\mathcal{N}}$ les inclusions à vérifier, on pose $\Omega_2 = \{$"on teste l'inclusion I_k à l'étape n $| \, k \in [\![1, \mathcal{N}]\!]$ et $n \in [\![1, N]\!]\}$. Au final, on a :

$$\Omega = \Omega_1 \times \Omega_2 \times \prod_{\mathbf{E} \in \mathcal{P}(\mathbf{Value})} \Omega_{\mathbf{E}}$$

On considère volontairement un espace d'évènements plus grand que nécessaire[5] pour simplifier la définition.

Dans les sections suivantes, étant donné que nous cherchons à établir une probabilité, nous essayons à chaque fois de nous ramener à des problèmes connus, notamment de tirages.

[5]Notamment en ce qui concerne les $\Omega_{\mathbf{E}}$.

FIG. 5 – Inclusion de **A** dans **B**

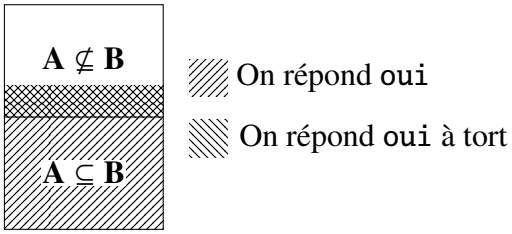
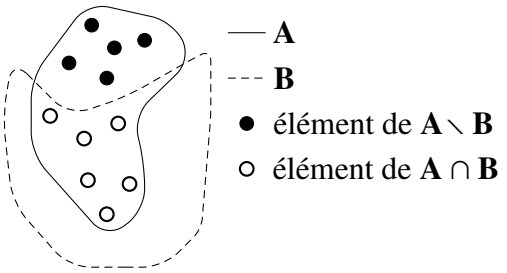


FIG. 6 – **A** est une urne, ses éléments des boules.



5.2 Vérification probabiliste d'une inclusion

Nous commençons par établir la probabilité d'accepter une inclusion à tort en la vérifiant partiellement.

Plus précisément, soient **A** et **B** deux ensembles. On souhaite vérifier que $\mathbf{A} \subseteq \mathbf{B}$. Pour cela, on choisit L fois aléatoirement un élément de **A** et on teste s'il appartient à **B**. Si, lors d'un essai où l'on prend un élément $a \in \mathbf{A}$, on a $a \notin \mathbf{B}$ alors on répond `non`. Sinon — c'est-à-dire si à chaque essai, on prend un élément $a \in \mathbf{A}$ tel que $a \in \mathbf{B}$ — on répond `oui`.

On remarque que si $\mathbf{A} \subseteq \mathbf{B}$ alors, on répond systématiquement `oui`. Autrement dit, on ne peut pas répondre `non` à tort. On cherche donc à évaluer la probabilité de répondre `oui` à tort. Autrement la probabilité de l'intersection de deux événements : "on répond `oui`" et "$\mathbf{A} \not\subseteq \mathbf{B}$" (voir figure 5). On calcule donc :

$$\begin{aligned}
\mathcal{P}_r(\text{“}\texttt{oui}\text{ à tort”}) &= \mathcal{P}_r(\text{“}\texttt{oui}\text{”} \cap \mathbf{A} \not\subseteq \mathbf{B}) && \text{(figure 5)} \\
&= \sum_{i=1}^{\#A} \mathcal{P}_r(\text{“}\texttt{oui}\text{”} \cap \#\mathbf{A} \setminus \mathbf{B} = i) && \text{(probabilités totales)} \\
&= \sum_{i=1}^{\#A} \mathcal{P}_r(\text{“}\texttt{oui}\text{”} | \#\mathbf{A} \setminus \mathbf{B} = i) \mathcal{P}_r((\#\mathbf{A} \setminus \mathbf{B}) = i) && \text{(probabilité conditionnelle)}
\end{aligned}$$

On va maintenant calculer une valeur explicite pour chaque membre du produit dans la somme : $\mathcal{P}_r(\text{“}\texttt{oui}\text{”} | (\#\mathbf{A} \setminus \mathbf{B}) = i)$ et $\mathcal{P}_r((\#\mathbf{A} \setminus \mathbf{B}) = i)$.

Pour le premier membre, nous montrons que le fait de tester L fois si un élément de **A** appartient à **B** sachant que i éléments de **A** n'appartiennent pas à **B** est équivalent à un tirage avec remise. Dans la figure 6, **A** est vu comme une urne, ses éléments sont vus comme des boules. La boule est noire si l'élément n'est pas dans **B** et blanche l'élément est dans **B**. De cette façon, on voit que tirer L fois un élément de **A** qui est aussi un élément de **B**, revient à tirer L fois une boule blanche dans une urne contenant #**A** boules dont $\#\mathbf{A} \setminus \mathbf{B}$ boules noires et $\#\mathbf{A} \cap \mathbf{B}$ boules blanches. Ce type de tirage suit la loi de probabilité binomiale :

$$\mathcal{P}_r(\text{“}\texttt{oui}\text{”} | \#\mathbf{A} \setminus \mathbf{B} = i) = \left(\frac{\#A - i}{\#A}\right)^L$$

Pour le second membre, on peut voir sur la figure 7 que pour déterminer $\mathcal{P}_r(\#\mathbf{A} \setminus \mathbf{B} = i)$, on peut se ramener à un tirage sans remise où **Value** est l'urne, #**B** le nombre de boules

FIG. 7 – **Add** est une urne, **B** l'ensemble des boules blanches, **A** le tirage mis dans une urne pour réaliser le tirage de la figure 6 .

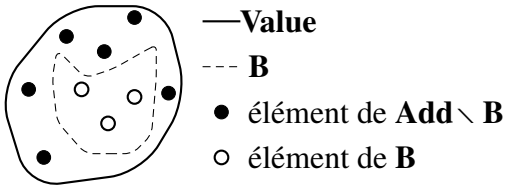
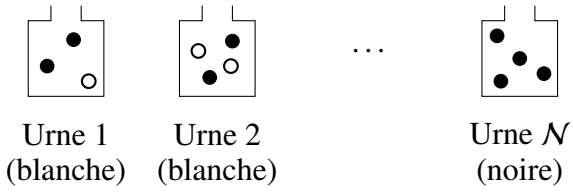


FIG. 8 – On choisit une urne aléatoirement, puis on effectue un tirage avec remise dessus.



blanches, #**A** le nombre de boules tirées, et i le nombre de boules noires tirées. Ce type de tirage suit la loi de probabilité hypergéométrique :

$$\mathcal{P}_r(\#\mathbf{A} \setminus \mathbf{B} = i) = \frac{C^i_{\#\mathbf{Add}-\#\mathbf{B}}\, C^{\#\mathbf{A}-i}_{\#\mathbf{B}}}{C^{\#\mathbf{A}}_{\#\mathbf{Add}}}$$

Au final, on a :

$$\mathcal{P}_r(\text{“}\texttt{oui}\text{ à tort”}) = \sum_{i=1}^{\#\mathbf{A}} \left(\frac{\#\mathbf{A} - i}{\#\mathbf{A}}\right)^L \frac{C^i_{\#\mathbf{Value}-\#\mathbf{B}}\, C^{\#\mathbf{A}-i}_{\#\mathbf{B}}}{C^{\#\mathbf{A}}_{\#\mathbf{Add}}}$$

Nous sommes donc en mesure d'estimer la probabilité de croire qu'une inclusion est vraie à tort. Nous allons maintenant calculer la probabilité de croire que l'ensemble d'un certificat est vrai à tort.

5.3 Vérification de l'ensemble du résultat

On souhaite tester N fois une inclusion au hasard parmi les $\mathcal{N}$ inclusions proposées. On voit sur la figure 8 que cela à revient à choisir au hasard N fois une urne parmi $\mathcal{N}$. Les urnes contiennent des boules noires ou blanches, et sont numérotés. On ne connaît pas le nombre d'urnes ne contenant que des boules noires (correspondant aux inclusions vraies) et que l'on appelle « urnes noires ». On connait pas non plus le nombre d'urnes contenant au moins une boule blanche (correspondant aux inclusions fausses) et que l'on appelle « urnes blanches ». En revanche, on sait calculer la probabilité d'une urne i d'être noire. De plus, on connaît la probabilité de ne tirer que des boules noires d'une urne blanche (cela a été calculé en section 5.2).

On prend comme hypothèse que les N tirages sont indépendants. Nous verrons dans la section 5.4 que cette hypothèse est fausse en toute généralité mais permet de simplifier la formule. De plus nous verrons dans la section 6 que les résultats expérimentaux sont proches des résultats théoriques faisant cette hypothèse.

Afin de calculer la probabilité qu'à l'algorithme de se tromper on calcule d'abord la probabilité de juger l'inclusion tirée vraie. L'hypothèse d'indépendance permet alors de déduire rapidement la probabilité de juger toutes les inclusions tirées vraies. En terme de tirage, cela signifie que l'on va d'abord calculer la probabilité de considérer une urne choisie noire, puis en déduire la probabilité de croire toutes les urnes noires.

On commence donc en calculant la probabilité P qu'une urnes tirée soit vue noire :

$$\begin{aligned}
P &= \mathcal{P}_r(\text{“urne vue noire”}) \\
&= \sum_{k=1}^{\mathcal{N}} \mathcal{P}_r(\text{“urne crue noire”}|\text{“urne n}^{\text{o}}\ k\ \text{tirée”})\mathcal{P}_r(\text{“urne n}^{\text{o}}\ k\ \text{tirée”}) \\
&\qquad\qquad \text{d'après la formule de probabilités totales} \\
&= \sum_{k=1}^{\mathcal{N}} \mathcal{P}_r(\text{“urne noire” } \cup \text{ “urne blanche crue noire”}|\text{“urne n}^{\text{o}}\ k\ \text{tirée”})\frac{1}{\mathcal{N}} \\
&\qquad\qquad \text{car } \mathcal{P}_r(\text{“urne n}^{\text{o}}\ k\ \text{tirée”}) = \tfrac{1}{\mathcal{N}} \\
&= \frac{1}{\mathcal{N}} \sum_{k=1}^{\mathcal{N}} \mathcal{P}_r(\text{“urne noire”}|\text{“urne n}^{\text{o}}\ k\text{”}) + \mathcal{P}_r(\text{“urne blanche crue noire”}|\text{“urne n}^{\text{o}}\ k\text{”}) \\
&\qquad\qquad \text{car les deux évènements sont disjoints}
\end{aligned}$$

Or, « urne noire » signifie que l'inclusion est vraie donc si l'urne k correspond à l'équation $\mathbf{A}_k \subseteq \mathbf{B}_k$, on a $\mathcal{P}_r(\text{“urne noire”}|\text{“urne n}^{\text{o}}\ k\text{”} = \mathcal{P}_r(\mathbf{A}_k \setminus \mathbf{B}_k = 0)$. C'est un cas particulier du problème de la figure 6 pour $i = 0$. On a donc $\mathcal{P}_r(\text{“urne noire”}|\text{“urne n}^{\text{o}}\ \text{k”}) = C^{\#\mathbf{A}_k}_{\#\mathbf{B}_k}/C^{\#\mathbf{A}_k}_{\#\mathbf{Add}}$. De plus, « croire une urne blanche noire » correspond à accepter une inclusion à tort, cela a déjà été calculé dans la section 6, on a donc :

$$\mathcal{P}_r(\text{“urne blanche crue noire”}|\text{“urne n}^{\text{o}}\ \text{k”}) = \sum_{i=1}^{\#\mathbf{A}_k} \left(\frac{\#\mathbf{A}_k - i}{\#\mathbf{A}_k}\right)^L \frac{C^i_{\#\mathbf{Add}-\#\mathbf{B}_k} C^{\#\mathbf{A}_k - i}_{\#\mathbf{B}_k}}{C^{\#\mathbf{A}_k}_{\#\mathbf{Add}}}$$

Enfin comme $C^{\#\mathbf{A}_k}_{\#\mathbf{B}_k}/C^{\#\mathbf{A}_k}_{\#\mathbf{Add}}$ est un cas particulier de la somme pour $i = 0$, on peut regrouper les deux cas. On a au final :

$$P = \mathcal{P}_r(\text{“urne crue noire”}) = \frac{1}{\mathcal{N}} \sum_{k=1}^{\mathcal{N}} \sum_{i=0}^{\#\mathbf{A}_k} \left(\frac{\#\mathbf{A}_k - i}{\#\mathbf{A}_k}\right)^L \frac{C^i_{\#\mathbf{Add}-\#\mathbf{B}_k} C^{\#\mathbf{A}_k - i}_{\#\mathbf{B}_k}}{C^{\#\mathbf{A}_k}_{\#\mathbf{Add}}}$$

Sous l'hypothèse que les tirages sont indépendants, la probabilité d'annoncer toutes les inclusions vraies est le produit des probabilités de croire l'urne choisie noire N fois. Autrement dit :

$$\mathcal{P}_r(\text{“annoncer inclusions vraies”}) = \left(\frac{1}{\mathcal{N}} \sum_{k=1}^{\mathcal{N}} \sum_{i=0}^{\#\mathbf{A}_k} \left(\frac{\#\mathbf{A}_k - i}{\#\mathbf{A}_k}\right)^L \frac{C^i_{\#\mathbf{Add}-\#\mathbf{B}_k} C^{\#\mathbf{A}_k - i}_{\#\mathbf{B}_k}}{C^{\#\mathbf{A}_k}_{\#\mathbf{Add}}}\right)^N$$

On souhaite connaître la probabilité d'annoncer toutes inclusions vraies à tort (puisqu'annoncer toutes inclusions vraies à raison est un comportement souhaité). Or d'après la loi des probabilités totales :

$$\begin{aligned}
\mathcal{P}_r(\text{“annoncer toutes inclusions vraies”}) = {} & \mathcal{P}_r(\text{“annoncer inclusions vraies à tort”}) + \\
& \mathcal{P}_r(\text{“annoncer inclusions vraies à raison”})
\end{aligned}$$

d'où

$$\begin{aligned}
\mathcal{P}_r(\text{“annoncer inclusions vraies à tort”}) = {} & \mathcal{P}_r(\text{“annoncer inclusions vraies”}) - \\
& \mathcal{P}_r(\text{“annoncer inclusions vraies à raison”})
\end{aligned}$$

La probabilité d'annoncer toutes les inclusions vraies a été calculée. La probabilité d'annoncer toutes les inclusions vraies à raison est simplement la probabilité que toutes les inclusions soient effectivement vraies. En effet, lorsque toutes les inclusions sont vraies,

l'algorithme annonce nécessairement qu'elles le sont. On a donc :

$$\begin{aligned}\mathcal{P}_r(\text{“annoncer à raison”}) &= \mathcal{P}_r(\text{“toutes les inclusions sont vraies”})\\ &= \prod_{k=1}^{\mathcal{N}} \mathcal{P}_r(\text{“urne noire”}|\text{“urne n}^{\text{o}}\text{ k”})\\ &= \prod_{k=1}^{\mathcal{N}} \frac{C_{\#\mathbf{B}_k}^{\#\mathbf{A}_k}}{C_{\#\mathbf{Add}}^{\#\mathbf{A}_k}}\end{aligned}$$

Au final on obtient la valeur théorique suivante :

$$\mathcal{P}_r(\text{“annoncer vrai à tort”}) = \left(\frac{1}{\mathcal{N}}\sum_{k=1}^{\mathcal{N}}\sum_{i=0}^{\#\mathbf{A}_k}\left(\frac{\#\mathbf{A}_k - i}{\#\mathbf{A}_k}\right)^L \frac{C_{\#\mathbf{Add}-\#\mathbf{B}_k}^{i} C_{\#\mathbf{B}_k}^{\#\mathbf{A}_k - i}}{C_{\#\mathbf{Add}}^{\#\mathbf{A}_k}}\right)^N - \prod_{k=1}^{\mathcal{N}} \frac{C_{\#\mathbf{B}_k}^{\#\mathbf{A}_k}}{C_{\#\mathbf{Add}}^{\#\mathbf{A}_k}} \quad (3)$$

On constate que comme on l'a expliqué intuitivement, le nombre d'inclusions testées N agit de manière plus globale que le nombre d'appartenances testées par inclusion L. Il est donc plus intéressant de prendre $L = 1$ et de maximiser N.

5.4 Indépendance des inclusions

L'un des reproches qui peut être fait au résultat précédent est qu'il est obtenu sous l'hypothèse que les inclusions sont indépendantes. Cette hypothèse est fausse en général. En effet, soit les inclusions à tester sont $\mathbf{A} \subseteq \mathbf{B}$, $\mathbf{B} \subseteq \mathbf{C}$ et $\mathbf{A} \subseteq \mathbf{C}$. Clairement, $\mathcal{P}_r(\mathbf{A} \subseteq \mathbf{C} \,|\, \mathbf{A} \subseteq \mathbf{B} \,\&\, \mathbf{B} \subseteq \mathbf{C}) = 1$.

Nous allons donc essayer de calculer la probabilité qu'une inclusion soit vraie sachant qu'une autre l'est. Formellement, soient $\mathbf{A_1}, \mathbf{B_1}, \mathbf{A_2}, \mathbf{B_2}$ des ensembles. On souhaite calculer $\mathcal{P}_r(\mathbf{A_2} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1})$. Pour cela, on commence par séparer le problème en deux problèmes plus petits en partitionnant l'événement $\mathbf{A_2} \subseteq \mathbf{B_2}$:

$$\mathcal{P}_r(\mathbf{A_2} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1}) = \mathcal{P}_r(\mathbf{A_2} \setminus \mathbf{A_1} \subseteq \mathbf{B_2} \,\&\, \mathbf{A_2} \cap \mathbf{A_1} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1})$$

Or, par application des probabilités conditionnelles, on peut séparer le second membre en un produit de deux facteurs :

$$\mathcal{P}_r(\mathbf{A_2} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1}) = \mathcal{P}_r(\mathbf{A_2} \setminus \mathbf{A_1} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1} \,\&\, \mathbf{A_2} \cap \mathbf{A_1} \subseteq \mathbf{B_2}) \mathcal{P}_r(\mathbf{A_2} \cap \mathbf{A_1} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1}) \quad (4)$$

On va maintenant simplifier le premier facteur de l'équation 4, à savoir : $\mathcal{P}_r(\mathbf{A_2} \setminus \mathbf{A_1} \subseteq \mathbf{B_2}|\mathbf{A_1} \subseteq \mathbf{B_1} \,\&\, \mathbf{A_2} \cap \mathbf{A_1} \subseteq \mathbf{B_2})$. On remarque d'abord que les événements $\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2$ et $\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1$ sont équivalents. On a donc :

$$\mathcal{P}_r(\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2|\mathbf{A}_1 \subseteq \mathbf{B}_1 \,\&\, \mathbf{A}_2 \cap \mathbf{A}_1 \subseteq \mathbf{B}_2) = \mathcal{P}_r(\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1|\mathbf{A}_1 \subseteq \mathbf{B}_1 \,\&\, \mathbf{A}_2 \cap \mathbf{A}_1 \subseteq \mathbf{B}_2)$$

Or, on pose l'hypothèse que les événements $\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1$ et $\mathbf{A}_1 \subseteq \mathbf{B}_1$ sont indépendants. Cela vient du fait que $(\mathbf{A}_2 \setminus \mathbf{A}_1) \cap \mathbf{A}_1 = (\mathbf{B}_2 \setminus \mathbf{A}_1) \cap \mathbf{A}_1 = \varnothing$. Autrement dit, $\mathbf{A}_1$ n'a rien à voir avec ces ensembles ; avoir un propriété sur $\mathbf{A}_1$ ne nous donne donc pas d'information pertinente. De même, les événements $\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1$ et $\mathbf{A}_2 \cap \mathbf{A}_1 \subseteq \mathbf{B}_2$ sont supposés indépendants. Finalement, on a simplement :

$$\mathcal{P}_r(\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2|\mathbf{A}_1 \subseteq \mathbf{B}_1 \,\&\, \mathbf{A}_2 \cap \mathbf{A}_1 \subseteq \mathbf{B}_2) = \mathcal{P}_r(\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1)$$

Après simplification, l'équation 4 devient :

$$\mathcal{P}_r(\mathbf{A}_2 \subseteq \mathbf{B}_2|\mathbf{A}_1 \subseteq \mathbf{B}_1) = \mathcal{P}_r(\mathbf{A}_2 \setminus \mathbf{A}_1 \subseteq \mathbf{B}_2 \setminus \mathbf{A}_1)\, \mathcal{P}_r(\mathbf{A}_2 \cap \mathbf{A}_1 \subseteq \mathbf{B}_2|\mathbf{A}_1 \subseteq \mathbf{B}_1) \quad (5)$$

Afin de pouvoir continuer les calculs, on observe que chacun des tests d'inclusion équivaut à la synthèse d'un ou plusieurs tests d'appartenance. La propriété « $\mathbf{A} \subseteq \mathbf{B}$ » est équivalente à la propriété « soient $a_1, \ldots, a_m$ les éléments de $\mathbf{A}$, on a $a_1 \in \mathbf{B}, \ldots, a_m \in \mathbf{B}$ ». Considérons donc les $\mathcal{M}$ propriétés d'appartenance correspondant aux $\mathcal{N}$ propriétés d'inclusion [6]. La probabilité conditionnelle ci-dessus s'applique : une propriété d'appartenance n'étant qu'une propriété d'inclusion dont l'ensemble inclus est réduit à un élément. Dans le cas particulier des appartenances, elle devient ceci :

$$\mathcal{P}_r(a_2 \in B_2 | a_1 \in B_1) = \mathcal{P}_r(\{a_2\} \setminus \{a_1\} \subseteq \mathbf{B}_2 \setminus \{a_1\}) \mathcal{P}_r(\{a_2\} \cap \{a_1\} \subseteq \mathbf{B}_2 | a_1 \in \mathbf{B}_1)$$

On remarque que l'un des deux facteurs est nécessairement trivial. En effet, soit $\{a_2\} \setminus \{a_1\} = \emptyset$, soit $\{a_2\} \cap \{a_1\} = \emptyset$.

On va maintenant procéder à une disjonction de cas :

- Si $\mathbf{B}_1 = \mathbf{B}_2$, deux cas se présentent :
  - Si $a_1 = a_2$ alors le problème est trivial :

$$\mathcal{P}_r(a_2 \in \mathbf{B}_2 | a_1 \in \mathbf{B}_1) = \mathcal{P}_r(a_1 \in \mathbf{B}_1 | a_1 \in \mathbf{B}_1) = 1$$

  - Si $a_1 \neq a_2$:

$$\mathcal{P}_r(a_2 \in \mathbf{B}_2 | a_1 \in \mathbf{B}_1) = \mathcal{P}_r(\{a_2\} \subseteq \mathbf{B}_2 \setminus \{a_1\}) \mathcal{P}_r(\emptyset \subseteq \mathbf{B}_2 | a_1 \in \mathbf{B}_1) = \mathcal{P}_r(a_2 \in \mathbf{B}_2 \setminus \{a_1\})$$

- si $\mathbf{B}_1 \neq \mathbf{B}_2$, comme aucune hypothèse n'est faite entre $\mathbf{B}_1$ et $\mathbf{B}_2$, on suppose qu'il y a indépendance :

$$\mathcal{P}_r(a_2 \in \mathbf{B}_2 | a_1 \in \mathbf{B}_1) = \mathcal{P}_r(a_2 \in \mathbf{B}_2)$$

Au final, nous pouvons prendre en compte la dépendance entre les différentes propriétés. Il est possible de calculer une valeur théorique semblable affinant le résultat de l'équation 3 en prenant en compte les dépendances. Pour cela, tout comme dans cette section, il faut séparer les propriétés d'inclusion en propriétés « atomiques » d'appartenance. Il faut alors estimer la probabilité de tester chaque sous ensemble $\mathbf{E}$ d'appartenance (tirage). On peut ensuite établir la probabilité que l'ensemble des appartenances est vérifié par le certificat sachant que les appartenances de $\mathbf{E}$ sont vérifiés (grâce aux résultats de cette section). Enfin, il suffit de regrouper.

Nous ne le faisons pas ici car l'équation 3 possède une erreur faible en pratique. Néanmoins, il nous a paru important de s'assurer que l'on peut alléger les hypothèses si cela est nécessaire pour de futurs travaux.

6 Implémentation et résultats expérimentaux

Dans la section 5, on a défini un algorithme de Monte-Carlo de vérification de certificats. Nous avons réalisé une implémentation de cet algorithme. Dans la section 6.1, nous discutons des choix d'implémentation et notamment ce que cela implique en termes de complexité temporelle et spatiale. Dans la section 6.2 nous donnons quelques résultats expérimentaux de notre implémentation.

6.1 Implémentation

Nous avons implémenté notre algorithme en `caml`. Dans une première approche nous avons choisi de représenter les valeurs (**Value**) par des entiers et les ensembles par des tableaux triés. De cette façon, nous obtenons une complexité spatiale dans le cas le pire

[6] Numéros d'indice choisis arbitrairement.

en $O(\mathcal{KL})$ où $\mathcal{K}$ est le nombre d'ensembles et $\mathcal{L}$ la taille maximale des ensembles. La complexité temporelle dans le cas le pire est elle en $O(NL(\log_2 \mathcal{L}))$ puisque nous réalisons une recherche dichotomique. Enfin le nombre de bits aléatoires générés est de $\log_2 \mathcal{K}$ pour choisir aléatoirement un ensemble et au pire $\log_2 \mathcal{L}$ pour choisir aléatoirement un élément dans un ensemble. Le nombre de bits aléatoires générés est donc de $N \log_2 \mathcal{K} + NL \log_2 \mathcal{L}$ dans le cas le pire. On remarque que si $L = 1$ on obtient $N(\log_2 \mathcal{KL})$ bit générés.

Pour résumer, comme $\mathcal{K}$ correspond aux inclusions qui correspondent aux instructions et $\mathcal{L}$ correspond aux valeurs qui correspondent aux adresses donc aux variables. On a une complexité spatiale linéaire en le nombre d'instructions et en le nombre de variables. La complexité temporelle est logarithmique en le nombre de variables et linéaire en chacun des paramètres de l'algorithme. Enfin, on génère un nombre de bits logarithmique en le nombre d'inclusion et de variables (réunis) et linéaire en chacun des paramètres L et N de l'algorithme.

6.2 Résultats expérimentaux

Lorsque l'algorithme a été implémenté, nous avons réalisé des tests en deux temps. D'abord, nous avons testé l'algorithme avec des certificats générés aléatoirement que nous avons comparés avec nos valeurs théoriques (équation 3). Ensuite, nous avons essayé d'affiner le comportement du producteur. En effet, des certificats générés aléatoirement, correspondent à un producteur qui ne réaliserait pas l'analyse de son logiciel. Un producteur pourrait, au contraire réaliser l'analyse, s'apercevoir qu'elle ne garantit pas que son logiciel se comporte bien et donc modifier le résultat de l'analyse pour camoufler cela. Quel que soit le cas considéré les équations ont toujours été générée aléatoirement et n'ont jamais correspondu à un vrai programme.

Dans un premier temps, pour des certificats réalisés aléatoirement, nous avons réalisé des tests pour des tailles de l'ensemble des valeurs (**Add** = **Value**) allant de 1 à 11 (valeurs qui peuvent correspondre à une fonction d'un programme), un nombre d'inclusions vérifiées allant de 1 à 11 et un nombre d'appartenances vérifiées par inclusion allant de 1 à 11. À chaque triplet de valeur, nous avons lancé 1 000 000 de fois l'algorithme et compté le nombre de fois qu'il obtenait un résultat erroné.

De ces tests ressortent trois résultats principaux. Le premier est que la probabilité d'erreur est bien exponentiellement décroissante (l'échelle des ordonnées de la figure 9 est logarithmique).

Le deuxième résultat (figure 9a) est que comme prévu, le nombre de test d'appartenances vérifiées par inclusion L a une influence moins grande que le nombre d'inclusions vérifiées N. En effet, la figure 9a compare les courbes pour $L = 1$ et $L = 11$. On voit alors, par exemple que pour $L = 1$ et $N = 9$ (9 tests d'appartenance au total), le taux d'erreur est de l'ordre de 1 pour 1 000 000, alors que pour $L = 11$ et $N = 3$ (33 tests d'appartenance au total), le taux d'erreur est de l'ordre de 1 pour 1 000.

Le troisième résultat (figure 9b) est que bien que faisant une hypothèse erronée en général, la formule théorique fournit une valeur proche des résultats expérimentaux.

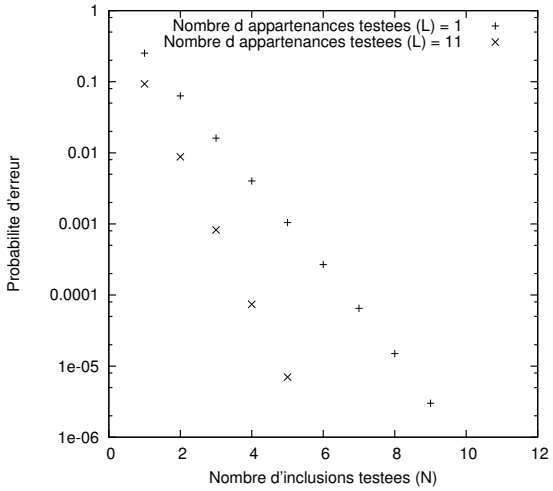
Dans un deuxième temps, pour un vrai certificat légèrement modifié, nous avons établi un certain nombre de manières dont la modification peut être faite. D'abord, des valeurs arbitraires peuvent être retirées d'ensembles arbitraires, ensuite certaines valeurs peuvent être tabou et retirées de tous les ensembles auxquels elles appartiennent.

Dans le premier cas, (figure 10) nous avons réalisé des tests pour avec un ensembles des valeurs (**Add** = **Value**) de taille 100[7]. Nous avons réalisé des tests pour un nombre d'erreurs allant de 1 à 101 avec un pas de 10 et un nombre d'inclusions vérifiées (N) allant

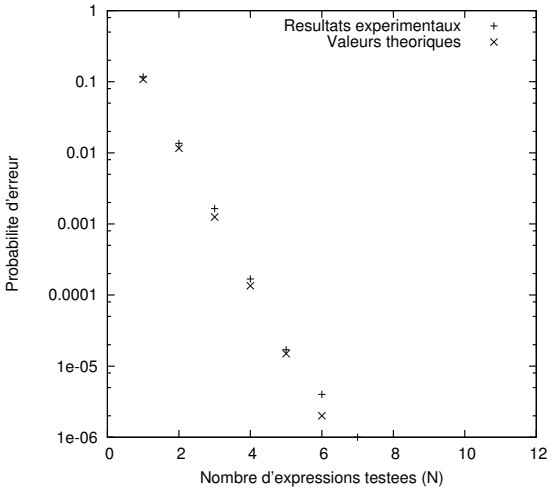
[7]Ceci car si l'ensemble des valeurs est trop petit, 1 erreur est déjà important

FIG. 9 – Résultats expérimentaux pour des certificats générés aléatoirement.

(a) Comparaison de l'influences du nombre d'inclusions testées et du nombre d'appartenances testées par inclusions (#**Add** = 11).



(b) Comparaison des résultats expérimentaux et des résultats théoriques (#**Add** = 11). Nombre d'appartenances testées (L) = 5



de 1 à 101 avec un pas de 10. Le nombre d'appartenances vérifiées par inclusion (L) était fixé à 1. Nous avons lancé 1000 fois l'algorithme pour chaque paire de paramètre.

Nous avons observé que pour une seule erreur la convergence est lente (figure 10). En effet avec 101 tests d'inclusion, le taux d'acceptation erronée est de l'ordre de 0.5. Cependant, il ne faut pas perdre de vue que le certificat est bien plus grand que l'ensemble des valeurs. En réalisant 101 tests d'inclusion, on réalise donc un nombre de tests très inférieur au nombre d'appartenance total à vérifier. On n'a pas fait plus de tests car les temps de calcul devenaient trop long pour lancer assez souvent l'algorithme. Afin de résoudre ce problème nous proposons dans les ouvertures (section 7) une implémentation ayant une meilleure complexité temporelle.

7 Ouvertures

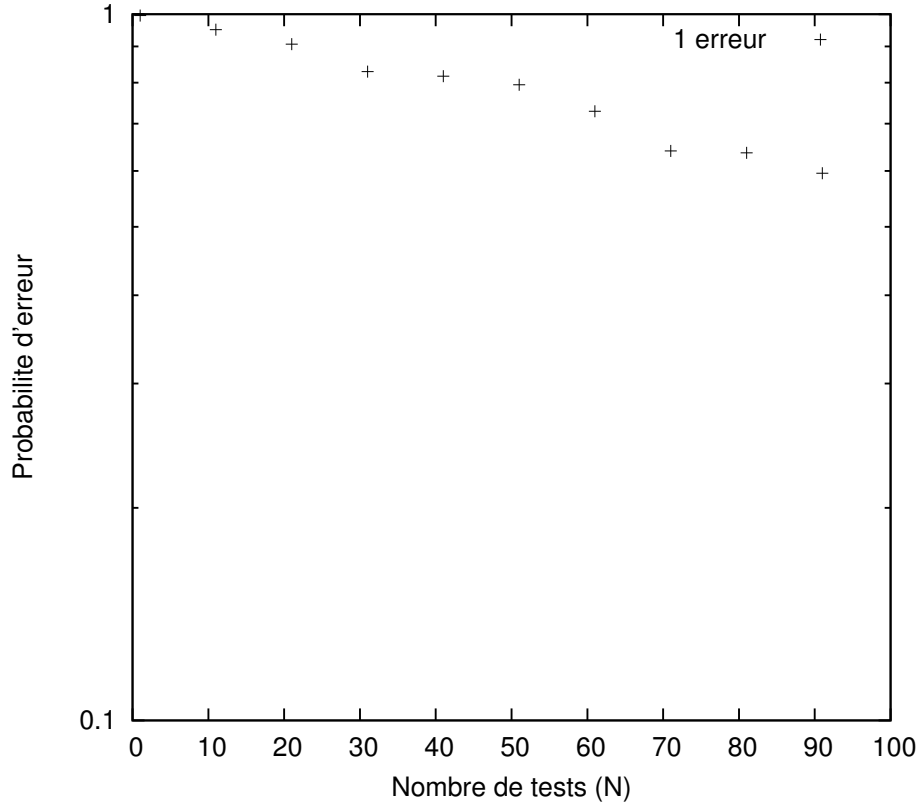
Notre travail ayant été présenté, nous donnons des pistes qui peuvent être suivies pour le généraliser ou l'améliorer. La section 7.1, traite des martingales, un outil de théorie de probabilité puissant et qui pourrait permettre de donner un cadre théorique général à la vérification probabiliste de certificat (plutôt que de traiter l'exemple d'une analyse spécifique). La section 7.2 propose, quant à elle, des améliorations possibles de l'implémentation de l'algorithme.

7.1 L'utilisation de martingales

Dans l'optique de généraliser le problème, nous allons montrer que l'on peut modéliser ledit problème par une surmartingale. L'algorithme choisi pour répondre au problème correspond alors à *la stratégie choisie pour optimiser la vitesse de convergence de la surmartingale*. L'intérêt d'une telle modélisation est d'avoir un cadre indépendant d'un certain nombre d'hypothèses et donc de pouvoir éventuellement obtenir des formules qui en sont indépendantes. On pourra alors savoir sous quelles hypothèses un algorithme est meilleur qu'un autre.

D'abord on explique brièvement ce qu'est une martingale et quelle est la théorie sous-

FIG. 10 – Résultats expérimentaux pour des certificats presque vrais.



jacente (section 7.1.1). Pour une définition complète, on pourra se référer aux sections 10 et 11 du cours IPPA ([Gal]). Pour une définition plus intuitive, on pourra lire la section 4.4 de *Randomized Algorithms* ([MR95]) . Ensuite, on montre que le problème peut se modéliser à l'aide d'une martingale (section 7.1.2).

7.1.1 Définition d'une martingale

Nous commençons donc par définir ce sur quoi toute la théorie des martingales se base : le conditionnement et les filtrations. Pour cela on doit définir formellement l'espace des probabilités. Nous considérons donc un univers Ω, et une tribu $\mathcal{F}$ sur Ω sur laquelle sera définie la mesure de probabilité $\mathcal{P}_r$. Soit X une variable aléatoire sur Ω et F une sous-tribu de $\mathcal{F}$, on va définir l'espérance conditionnelle de X par rapport à F. Intuitivement, il s'agit de la variable aléatoire qui est la meilleure approximation de X sur F. Formellement, elle est définie ainsi :

$$\mathcal{E}(X \mid F) : \mathbf{F} \in F \mapsto \begin{cases} \frac{\mathcal{E}(X\mathbf{1}_{\mathbf{F}})}{\mathcal{P}_r(\mathbf{F})} & si\ \mathcal{P}_r(\mathbf{F}) \neq 0 \\ 0 & sinon \end{cases}$$

Où $\mathbf{1}_{\mathbf{F}}$ est la fonction qui vaut 1 sur $\mathbf{F}$ et 0 ailleurs. Une propriété importante de l'espérance conditionnelle est qu'elle est linéaire ($\mathcal{E}(aX + bY \mid F) = a\mathcal{E}(X \mid F) + b\mathcal{E}(X \mid F)$). De plus $\mathcal{E}(X \mid F)$, étant définie comme une variable aléatoire sur F est évidemment, F-mesurable [8]. De même, si X est F-mesurable, sa meilleure approximation sur F est elle-même puisque X peut-être définie sur F ; autrement dit, $\mathcal{E}(X \mid F) = X$.

Par exemple considérons un lancé de dé équilibré. On a $\Omega = \{1, 2, 3, 4, 5, 6\}$, $\mathcal{F} = \mathcal{P}(\Omega)$ et $\mathcal{P}_r : \mathbf{F} \in \mathcal{F} \mapsto (\#\mathbf{F})/6$. Soit X la variable aléatoire représentant la valeur du dé et F la

[8] Pour rappel : soit F une tribu, une variable X est $F-$ mesurable si et seulement si elle peut être définie comme une variable aléatoire sur F.

tribu engendrée par {pair, impair}. On a $\mathcal{E}(X \mid F)(\text{pair}) = 4$, c'est à dire l'espérance de X sur les valeurs paires, et $\mathcal{E}(X \mid F)(\text{impair}) = 3$, c'est à dire l'espérance de X sur les valeurs impaires.

On définit ensuite une filtration d'une tribu comme une approximation de plus en plus précise de celle-ci. Formellement, une filtration de $\mathcal{F}$ notée $(F_n)_{n\in\mathbf{N}}$ est une suite de sous-tribus de $\mathcal{F}$ telle que $\forall n \in \mathbf{N},\ F_n \subseteq F_{n+1}$.

Par exemple, considérons Ω = { population française } et $\mathcal{F} = \mathcal{P}(\Omega)$. Une filtration peut être la suite de tribus engendrée par Ω (F_0), { hommes, femmes } (F_1), { hommes de moins de 15 ans, hommes de plus de 15 ans, femmes de moins de 15 ans, femmes de plus de 15 ans} (F_2). En conditionnant une variable aléatoire par une filtration, on obtient alors une suite variables aléatoires de plus en plus précises. Par exemple soit Y la variable aléatoire qui donne la masse d'un français choisi au hasard. On a alors $\mathcal{E}(Y|F_0)$ qui traduit la masse moyenne des Français, $\mathcal{E}(Y|F_1)$ qui correspond à une variable donnant soit la masse moyenne des hommes en France soit la masse moyenne des femmes en France, et $\mathcal{E}(Y|F_2)$ qui ajoute la distinction entre les plus et moins de 15 ans.

Nous pouvons maintenant définir ce que sont les martingales. Le principe d'une martingale est de modéliser l'évolution d'une variable aléatoire « à mesure que nos connaissances augmenten ». On pose donc une filtration $(F_n)_{n\in\mathbf{N}}$ qui correspond à l'augmentation des connaissances. En effet, plus n grandit, plus F_n grandit et est fine, plus on peut faire des distinctions entre les événements. On pose alors une suite de variables aléatoires $(X_n)_{n\in\mathbf{N}}$ qui est la valeur que l'on souhaite estimer. On exige que pour tout n, X_n est F_n mesurable. Cela signifie que l'on connaît la valeur de X_n, si l'on a connaissance de F_n. Une surmartingale (qui le type de martingale qui nous intéresse) possède de plus la propriété suivante : $\forall n \in \mathbf{N},\ X_n \geq \mathcal{E}(X_n + 1 \mid F_n)$. Cela revient à dire qu'*en moyenne* la suite $(X_n)_{n\in\mathbf{N}}$ décroît.

Nous allons prendre l'exemple de l'estimation qu'à notre algorithme probabiliste de se tromper.

7.1.2 Modéliser le comportement de l'algorithme par une surmartingale

Pour modéliser le comportement de l'algorithme par une surmartingale nous définissons d'abord l'univers, puis la filtration, pour enfin définir la suite de variables aléatoires.

Afin de définir Ω nous considérons à nouveau les $\mathcal{M}$ appartenances, notées $A_1, \ldots, A_{\mathcal{M}}$, que l'algorithme peut vérifier. De plus, on partage l'algorithme de paramètres N et L en $N \times L$ étapes atomiques. Chacune de ces étapes correspond au test d'une appartenance. On pose alors :

$$\Omega = \prod_{\substack{i\in[\![1,\mathcal{M}]\!] \\ j\in[\![1,N\times L]\!]}} \{\text{appartenance } i \text{ vraie testée étape } j\} \times \{\text{appartenance } i \text{ fausse testée étape } j\}$$

Intuitivement on peut calculer sur Ω que chaque appartenance soit fausse ou non ainsi que les probabilités de tester chaque appartenance à chaque étape de l'algorithme.

Nous souhaitons modéliser le fait qu'au fur et à mesure que l'algorithme se déroule, nous connaissons les appartenances qui ont été testées ainsi que si elles sont vraies ou non. On définit alors $(F_n)_{n\in\mathbf{N}}$ inductivement. On pose $F_0 = \{\Omega, \varnothing\}$ c'est la tribu qui n'apporte aucun information sur X. La tribu F_{n+1}, par définition, est ensuite engendrée par l'ensemble suivant :

$$F_n \cup \bigcup_{i\in[\![1,\mathcal{M}]\!]} \{\text{appartenance } i \text{ vraie testée étape } n+1\} \cup \{\text{appartenance } i \text{ fausse testée étape } n+1\}$$

On définit enfin $(X_n)_{n\in\mathbf{N}}$ comme la probabilité de répondre oui à tort après l'étape n. On a bien la propriété que $\forall n, X_n$ est F_n-mesurable car cela revient à dire que si l'on sait

tout de ce qui a été testé jusqu'à l'étape n, on sait calculer la probabilité de se tromper après l'étape n. On remarque que X_0 correspond simplement à la probabilité de recevoir un certificat vrai.

Pour montrer que l'on a un surmartingale, on va montrer que la suite $(X_n)_{n\in\mathbf{N}}$ décroît :

$$\forall n, \forall \mathbf{F} \in F_n,\ X_n(\mathbf{F}) \geq (x \mapsto X_{n+1}(\mathbf{F} \wedge \{x\})) \tag{6}$$

Intuitivement, cette formule dit que la valeur de X_n quand le résultat des n premiers lancés $\mathbf{F}$ est connu est plus grande que toutes les valeurs possibles de X_{n+1} quand le résultat du $(n+1)-$ième lancé x sera connu. Cette propriété est plus forte que la propriété des surmartingales. En effet, si la suite décroît, *a fortiori* elle décroît *en moyenne*. Formellement, quand en prenant les espérances on obtient $\forall n, \forall \mathbf{F} \in F_n,\ X_n \geq \mathcal{E}(X_{n+1} \,|\, \mathbf{F})$. Autrement dit, $\forall n \in \mathbf{N},\ X_n \geq \mathcal{E}(X_{n+1} \,|\, F_n)$.

Pour montrer que la propriété 6 est vraie, on considère sans perte de généralité que c'est l'appartenance A_1 qui est testée à l'étape $n+1$[9]. Nous procédons alors à une disjonction de cas sur A_1.

premier cas A_1 est fausse. Dans ce cas on répond non. On ne peut donc pas répondre oui à tort : $X_{n+1}(\mathbf{F} \wedge \{x\}) = 0 \leq X_n(\mathbf{F})$ car $X_n \in [0,1]$ (évaluation de probabilité).

second cas A_1 est vraie. Dans ce cas nous procédons à une disjonction de cas sur $\mathbf{F}$. Si il existe « A_i est fausse » dans $\mathbf{F}$ alors $X_n = X_{n+1} = 0$. Ce cas n'est pas trés intéressant en pratique car l'algorithme s'est arrêté et a répondu non. Sinon, $\mathbf{F}$ est de la forme $\{A_{f_1}$ est vraie, $\ldots, A_{f_n}$ est vraie$\}$. On va maintenant expliciter X_n et X_{n+1}.

La variable aléatoire X_n représente la probabilité de dire oui à tort après n étapes. On a alors $\mathcal{P}_r(X_n \,|\, \mathbf{F}) = 1 - \mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F})$. En effet comme $\mathbf{F}$ ne contient que des appartenances vraies, on répond oui après n étapes, donc la probabilité de dire oui à tort est la probabilité que toutes les appartenances ne soient pas vérifiées.

De la même manière, la variable aléatoire X_{n+1} représentant la probabilité de dire oui à tort après $n+1$ étapes, on a $\mathcal{P}_r(X_n \,|\, \mathbf{F}) = 1 - \mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F} \wedge \{A_1 \text{ est vraie}\})$.

Au final l'inégalité $\mathcal{P}_r(X_n \,|\, \mathbf{F}) \geq \mathcal{P}_r(X_{n+1} \,|\, \mathbf{F})$ est équivalente à $\mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F}) \leq \mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F} \wedge \{A_1 \text{ est vraie}\})$. Or, soit $P = \mathcal{P}_r(\text{toutes appartenances vraies} \,|\, \mathbf{F})$, on a :

$$\begin{aligned}
P &= \mathcal{P}_r(A_1 \text{ vraie} \,\&\, \ldots \,\&\, A_{\mathcal{M}} \text{ vraie} \,|\, \mathbf{F}) \\
&= \mathcal{P}_r(A_1 \text{ vraie} \,|\, \mathbf{F}) \mathcal{P}_r(A_2 \text{ vraie} \,\&\, \ldots \,\&\, A_{\mathcal{M}} \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}\}) \\
&\qquad \text{(probabilités conditionnelles)} \\
&\leq \mathcal{P}_r(A_2 \text{ vraie} \,\&\, \ldots \,\&\, A_{\mathcal{M}} \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}\}) \\
&\qquad \text{car } \mathcal{P}_r(A_1 \text{ vraie} \,|\, \mathbf{F}) \leq 1 \\
&= \mathcal{P}_r(A_1 \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}) \mathcal{P}_r(A_2 \text{ vraie} \,\&\, \ldots \,\&\, A_{\mathcal{M}} \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}\} \wedge \{A_1 \text{ vraie}\}) \\
&\qquad \text{car } \mathcal{P}_r(A_1 \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}) = 1 \\
&= \mathcal{P}_r(A_1 \text{ vraie} \,\&\, \ldots \,\&\, A_{\mathcal{M}} \text{ vraie} \,|\, \mathbf{F} \wedge \{A_1 \text{ vraie}) \\
&\qquad \text{(probabilités conditionnelles)} \\
&= \mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F} \wedge \{A_1 \text{ est vraie}\})
\end{aligned}$$

Finalement, on a bien $\mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F}) \leq \mathcal{P}_r(\text{ toutes appartenances vraies} \,|\, \mathbf{F} \wedge \{A_1 \text{ est vraie}\})$.

□

[9] En effet, les appartenances sont numérotés arbitrairement.

En conclusion, on peut représenter la probabilité qu'a notre algorithme de se tromper par une surmartingale. Nous remarquons que noun n'avons pas utilisé d'hypothèse sur la dépendance entre les relations d'appartenance. De plus, notre preuve ne repose que sur le fait que l'algorithme teste des appartenances. La surmartingale que nous avons définie est donc plus générale que notre algorithme. D'abord, elle peut modéliser un algorithme différent de 1. Ensuite, on peut parfaitement la transposer à une analyse sensible au flot (ou **Label** $\neq \{.\}$, cf. section 4). Pour finir, un algorithme de vérification d'un certificat censé être une sémantique collectrice (section 3.4) peut aussi être modélisée par cette même surmartingale. En effet cela consiste à tester que des ensembles d'ensembles sont inclus les uns dans les autres. Ces tests peuvent être modélisés de manière atomique par plusieurs tests d'appartenance.

Un intérêt des martingales, outre un cadre général est l'existence d'outils mathématiques permettant de borner (voire calculer) la vitesse de convergence. Dans le cas de notre martingale, il est évident que $(X_n)_{n\in\mathbf{N}}$ converge vers 0 après un nombre inifini d'étapes (car il y a un nombre fini d'appartenance à vérifier). On peut cependant se demander à partir de quelle étape on a $X_n \leq 0.05$ par expemple. La théorie des martingales propose un outil pour cela : le temps d'arrêt. Intuitivement, le temps d'arrêt est une étape auquel on peut choisir de s'arrêter. Autrement dit, quand on atteint le temps d'arrêt, on le sait. Par exemple, « la dernière étape à laquelle on teste A_1 » n'est pas un temps d'arrêt car au moment où l'on atteint cette étape, on ne sait pas que l'on ne testera plus A_1. En revanche, « l'étape à partir de laquelle $X_n \leq 0.05$ » est un temps d'arrêt car X_n décroît. On peut définir la variable aléatoire qui donne le numéro de l'étape ou le temps d'arrêt advient. La théorie des martingales fournit des outils pour calculer la loi de cette variable aléatoire.

Maintenant que nous avons une possible amélioration de l'approche théorique, nous proposons une manière d'améliorer l'implémentation.

7.2 Amélioration de l'implémentation

Nous proposons ici deux améliorations de l'implémentation possibles, elles sont indépendantes. La première vise à améliorer le temps d'exécution de notre algorithme. Le seconde consiste à faire en sorte de pouvoir générer des équations qui correspondent à des vrais programmes.

En ce qui concerne la complexité temporelle, nous pensons l'améliorer en implémentant les ensembles sous forme tableaux de bits. Le bit n du tableau correspondrait à la valeur n, un bit à 0 signifierait que la valeur n'appartient pas à l'ensemble, un bit à 1 signifierait que la valeur appartient à l'ensemble. L'avantage de cette implémentation est que la complexité spatiale reste en $O(\mathcal{KL})$[10] alors que la complexité temporelle passe en $O(NL)$ puisque les accès aux tableaux se feront en temps constant. La difficulté de cette implémentation est de prendre efficacement un élément aléatoire dans un ensemble (étape 5 de l'algorithme, cf. 5.1).

À propos de la génération d'équations à partir de vrais programmes, nous étudions la possibilité de nous interfacer à `sawja`[11]. Cela nous permettrait alors d'analyser du `byte code java` ce qui est précisément le contexte dans lequel nous nous plaçons en section 1.

[10] La constante est à ce propos réduite puisque l'on n'a plus besoin que d'un bit par valeur plutôt que la taille d'un entier (32 bits le plus souvent).

[11] http ://sawja.inria.fr/

8 Conclusion

Pour conclure, nous avons, à notre connaissance, réalisé le premier algorithme de vérification d'un certificat par rapport à une analyse statique. Nous sommes, pour ce faire, placé dans un sous-contexte précis que nous maitrisions afin de pouvoir acquérir une meilleure connaissance de la problématique. De plus, nous avons implémenté notre solution pour ne pas perdre vue l'objectif applicatif que nous avons. L'implémentation a été testée et les premiers résultats sont encourageants bien que du travail puisse encore être fait en termes de complexité temporelle et de pertinences des équations testées.

De plus, nous sommes maintenant en mesure de proposer des axes de recherche plus généraux afin de ne plus avoir s'astreindre au sous-problème que nous nous sommes posé. Nous estimons que l'utilisation de martingales, notamment, permet d'aller dans ce sens.

Références

[Gal] Jean-François Le Gall. Intégration, probabilités, processus aléatoires. www.math.u-psud.fr/~jflegall/IPPA2.pdf.

[GN03] Sumit Gulwani and Georges C. Necula. Discovering affine equalities using random interpretation. In 30$^{\text{th}}$ *Annual Symposium ACM on Principles of Programming Langages*, pages 74–84, 2003.

[GN04a] Sumit Gulwani and Georges C. Necula. Global value numbering using random interpretation. In 31$^{\text{st}}$ *Annual Symposium ACM on Principles of Programming Langages*, pages 342–352, 2004.

[GN04b] Sumit Gulwani and Georges C. Necula. A polynomial-time algorithm for global value numbering. In 11$^{\text{th}}$ *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227. Springer, 2004.

[GN05] Sumit Gulwani and Georges C. Necula. Precise interprocedural analysis using random interpretation. In 32$^{\text{nd}}$ *Annual Symposium ACM on Principles of Programming Langages*, 2005.

[Gul05] Sumit Gulwani. *Program analysis using random interpretation*. PhD thesis, University of California, Berkeley, 2005.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[NG11] Rupesh Nasre and Ramaswamy Govindarajan. Points-to analysis as a system of linear equations. In *Lecture Notes in Computer Science*, volume 6337, pages 422–438. Springer, 2011.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Lecture Notes in Computer Science*, volume 3780, pages 98–118. Springer, 2005.

[WL04] John Whaley and Monica S. Lam. Cloning-based context sensitive pointer alias analysis using binary decision diagrams. In *conference on Programming language Design and Implementation*, 2004.