Opacité des artefacts d'un système Workflow

Mohamadou Lamine Diouf

▶ To cite this version:

Mohamadou Lamine Diouf. Opacité des artefacts d'un système Workflow. Autre [cs.OH]. Université de Rennes; Université Cheikh Anta Diop (Dakar), 2014. Français. NNT : 2014REN1S061 . tel-01127287

HAL Id: tel-01127287
https://theses.hal.science/tel-01127287

Submitted on 7 Mar 2015

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

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

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

En Cotutelle Internationale avec

Université Cheikh Anta DIOP de Dakar, Sénégal

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES1

Mention : Informatique

École doctorale MATISSE

présentée par

Mohamadou Lamine DIOUF

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires
ISTIC

Opacité des artefacts d'un système Workflow

Thèse soutenue à Rennes le

devant le jury composé de :

Béatrice BOUCHOU MARKHOFF
Maître de conférences HDR à l'université de Tours / *Rapporteur*

Moussa LO
Chercheur à l'Université Gaston Berger de Saint-Louis / *Rapporteur*

Sophie PINCHINAT
Professeur à l'Université de Rennes 1 / *Examinateur*

Djiby SOW
Professeur à l'Université Cheikh Anta Diop de Dakar / *Examinateur*

François CHAROY
Professeur à l'Université de Lorraine à Nancy / *Examinateur*

Eric BADOUEL
Chargé de recherche INRIA / *Directeur de thèse*

Oumar DIANKHA
Professeur à l'Université Cheikh Anta Diop de Dakar / *Co-directeur de thèse*

Ou bien parais tel que tu es, ou bien sois tel que tu parais.
par Djalâl-od-Dîn Rûmî

Remerciements

Je remercie Béatrice Bouchou Markhoff, Maître de conférences HDR à l'université de Tours, et Moussa Lo, Chercheur à l'Université Gaston Berger de Saint-Louis, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Sophie Pinchinat, Professeur à l'Université de Rennes 1, Djiby Sow, Professeur à l'Université Cheikh Anta Diop de Dakar, et François Charoy, Professeur à l'Université de Lorraine à Nancy, d'avoir bien voulu juger ce travail.

Je remercie Oumar DIANKHA, Professeur à l'Université Cheikh Anta DIOP de Dakar, d'avoir bien voulu co-encadrer ma thèse.

Je remercie enfin Eric BADOUEL, Chargé de recherche Inria, qui a dirigé ma thèse .

Table des matières

Introduction **3**

1 Conformité d'un artefact **9**

2 Abstraction d'un artefact **19**

3 Opacité d'un artefact **25**

4 Implémentation **37**

Conclusion **67**

Annexe : Arithmétique de Presburger **69**

Bibliographie **79**

Introduction

La propriété centrale qui nous intéresse dans ce travail est celle d'*opacité* [HS04, BKMR08] dont nous donnons, dans le contexte de cette thèse, la présentation suivante. Soit un ensemble U —pour *univers*— contenant les objets qui nous intéressent. Dans le cas des systèmes à événements discrets il s'agit de l'ensemble des exécutions d'un système. Dans notre cas il s'agira de l'ensemble des dossiers manipulés par un système administratif. Ceux-ci sont représentés par les documents structurés —à la XML— conformes à un certain schéma (ou grammaire). Un observateur est représenté par sa fonction d'observation $\phi : U \to O$ dans laquelle O est l'ensemble des observations possibles et $\phi(x)$ est l'information que l'observateur a de l'objet x. Une propriété des objets peut être assimilée à l'ensemble des objets qui la vérifient, c'est-à-dire qu'il s'agit d'un ensemble $P \subseteq U$ et on dit que "x vérifie la propriété P" lorsque $x \in P$ et que "x ne vérifie pas la propriété P" si $x \notin P$. Une telle propriété P est *opaque* pour l'observateur si celui-ci ne peut pas déduire que l'objet qu'il observe satisfait la propriété sur la base de ce qu'il peut observer de celui-ci.

Formellement on peut exprimer que P est alors opaque vis-à-vis de ϕ lorsque

$$\forall x \in P \;\; \exists y \notin P \quad \phi(x) = \phi(y) \tag{1}$$

c'est-à-dire qu'il n'est pas possible de déduire qu'un objet vérifie la propriété ($x \in P$) sur la base de son observation $\phi(x)$ puisqu'il est en effet possible de trouver un autre élément y donnant lieu à la même observation ($\phi(x) = \phi(y)$) et qui néanmoins ne vérifie pas cette propriété ($y \notin P$). L'équation (1) peut se reformuler de façon ensembliste : la propriété P est opaque vis-à-vis de ϕ si, et seulement si,

$$\phi(P) \subseteq \phi(U \setminus P) \tag{2}$$

L'hypothèse sous-jacente à la notion d'opacité est que l'observateur a une parfaite connaissance du système : il connait l'ensemble U —dans notre cas la grammaire décrivant les documents—, il sait par ailleurs comment sa fonction d'observation ϕ est construite et il est en mesure de déterminer l'ensemble $\phi^{-1}(o)$ des objets qui donnent lieu à une observation donnée $o \in O$. De cette façon, la connaissance qu'il a d'un objet $x \in U$ est donnée par l'ensembe $\phi^{-1}(\phi(x))$ des

objets qui donnent lieu à la même observation que celui-ci. Ainsi il sera capable de déterminer qu'un objet x vérifie la propriété P lorsque

$$\phi^{-1}(\phi(x)) \subseteq P \tag{3}$$

De fait les formules —équivalentes— (1) et (2) peuvent se reformuler en disant que la condition (3) ne doit être vérifiée par aucun élément $x \in U$. Lorsque la relation (3) est vérifiée pour un certain objet x on dit que cet objet donne lieu à une *fuite d'information*, ce qui constitue une faille de sécurité dans le cas où la propriété qui a été déduite est une information sensible qu'on souhaitait garder secrète. L'élément x vérifiant la relation (3) est appelé un *témoin* de la fuite de l'information P. Il faut faire attention au fait que la non opacité d'une propriété P ne signifie pas qu'un observateur sera toujours dans la possibilité de déterminer si la propriété P est vraie ou non pour l'objet qu'il observe mais seulement qu'il existe des cas —donnés par les témoins— pour lesquels il est capable de déduire que la propriété est satisfaite. On doit noter par ailleurs que l'opacité d'une propriété P n'est pas équivalente à l'opacité de sa négation : il peut exister des témoins pour P sans qu'il existe des témoins pour $\neg P$. Cette asymétrie de l'opacité est particulièrement visible dans la formulation (2) de cette propriété.

Supposons qu'à chaque intervenant du système (identifié à sa fonction d'observation $\phi : U \to O$) soit attaché un certain nombre de propriétés $S_1, \cdots S_n \subseteq U$. Il s'agit de propriétés du système, appelées *secrets*, qui ne doivent pas être dévoilées à cet utilisateur. Typiquement certaines informations confidentielles doivent être accessibles à certains utilisateurs autorisés et être tenues secrètes pour les autres. Par exemple les coordonnées bancaires d'un client doivent être connues du service de comptabilité mais pas des autres services du système. Nous exigeons alors qu'aucun secret ne puisse être dévoilé : le système est *opaque* si pour chaque observateur $\phi : U \to O$ et pour chacun des secrets $S \subseteq U$ attachés à cet observateur on a $\phi(S) \subseteq \phi(U \setminus S)$, c'est-à-dire que le secret S est opaque vis-à-vis de ϕ.

Le problème de l'opacité a été jusqu'à présent étudié dans le contexte des systèmes à événements discrets et a servi à modéliser des problèmes de sécurité et/ou de confidentialité pour des systèmes informatiques ou des protocoles, voir e.g. [BKMR08, Lin11]. Nous reviendrons sur ces travaux lorsqu'il s'agira d'envisager des techniques pour assurer l'opacité d'un système en s'inspirant des techniques de monitoring [DJM09] ou de contrôle [BBB$^+$07] qui ont été développées dans ce contexte d'origine.

Notre travail est la première tentative d'appliquer ce concept d'opacité à des systèmes à flots de tâches centrés sur les données (*data-centric workflow systems* [CH09]). Il s'agit de systèmes distribués asynchrones dans lesquels la coordination des activités s'effectue par la transmission de documents structurés (à la XML) combinant structure logique, et données. Les éléments de l'univers de référence U sont donc des documents structurés, appelés *artefacts*, qui servent de support à

la réalisation d'une tâche : il s'agira par exemple du dossier médical d'un patient, d'un dossier administratif, d'un dossier pour le suivi d'une commande ... Au cours de ce traitement nous avons besoin d'invoquer un certain nombre de services auxquels nous serons ainsi amené à transmettre des informations extraites de ce dossier. Il y a plusieurs raisons qui conduisent à ne transmettre qu'une partie des informations contenues dans l'artefact. D'une part nous voulons éviter de surcharger un service par des informations qui ne sont pas utiles à la réalisation de la tâche qui lui incombe. D'autre part pour des raisons de confidentialité, et c'est ce point qui nous intéresse ici, certaines informations sensibles ne doivent pas être connues de tous. Enfin les services invoqués ont généralement été définis antérieurement et de façon par conséquent indépendante du workflow qui les invoque ; celui-ci doit donc extraire de l'artefact un document conforme à ce que le service est préparé à recevoir.

La discussion précédente va nous guider dans le choix des structures grammaticales (statuant de la conformité d'un document) ainsi que sur les fonctions d'observations $\phi : U \to O$ utilisées pour l'extraction des informations à transmettre à un service donné. Un autre critère important sur le choix de ces deux paramètres est de pouvoir aboutir à des algorithmes permettant de vérifier de manière effective si un système donné est opaque ou non. Un de nos principaux objectif est effectivement d'aboutir à une formalisation des notions de structure grammaticale des documents, de fonctions d'observation et de secrets qui garantisse la décision de l'opacité. Concrètement nous souhaitons obtenir un algorithme nous indiquant si un système est opaque ou non et qui dans la négative nous construit des témoins pour chacun des secrets qui peuvent être dévoilés.

Nous donnons quelques éléments sur la façon dont on peut modifier le système pour assurer son opacité. La première approche, qui s'inspire des techniques de contrôle des systèmes à événements discrets, permet de construire une nouvelle grammaire qui restreint les documents de la grammaire d'origine à ceux pour lesquels aucun secret n'est dévoilé. Cette technique peut avoir l'inconvénient d'écarter des documents que le concepteur du système aurait par ailleurs toutes les raisons de vouloir considérer comme parfaitement licites. Une méthode moins intrusive, qui s'inspire de techniques d'extensions conservatives de schémas XML, consiste au contraire à étendre la grammaire.

La thèse est organisée comme suit. Dans le chapitre 1, on présente les arbres

conformes à des grammaires d'arbres étendues, i.e., les parties droites de ces grammaires sont données par des expressions régulières. Les arbres conformes à ces grammaires représentent les artefacts. On s'interesse par la suite à la structure physique et logique de l'artefact ainsi qu' à son évaluation selon une algèbre. Un artefact (liste de documents) est conforme s'il appartient au langage de la gram-

maire et en particulier chacun des documents qui le constitue est bien formé. Une fois que la conformité de l'artefact est acquise on peut extraire de l'information en l'évaluant selon des algèbres de la grammaire. La notion d'algèbre sera rappelée dans ce chapitre, puis comparée avec différentes représentations à savoir les signatures multisortes et une représentation sous forme de productions.

Le chapitre 2 s'intéresse aux données contenues dans un artefact et de son observation ou abstraction vis-à-vis d'un utilisateur du système. On caractérise les arbres obtenus par observation, grâce à la fonction de projection. Ensuite on donne la relation principale qui existe entre la grammaire de départ et la grammaire projetée ou grammaire quotient qui reconnaît les projections des arbres conformes à la grammaire de départ.

Le chapitre 3 s'intéresse aux problèmes d'invocation de services web et d'opacité. Dans le premier cas les documents transmis doivent être reconnus conformes au service appelé. Dans les deux cas les documents transmis ou observables sont associés à un sous ensemble de sortes visibles, donc obtenus par une projection de la grammaire de départ. Malheureusement les langages qui caractérisent les projections des parties droites des règles de la grammaire quotient sont des langages algébriques. La grammaire d'arbres obtenues n'est donc pas algébrique (il faudrait que ses parties droites soient des langages réguliers). Pour contourner cette difficulté on introduit la notion de conformité modulo commutations pour rester dans le cadre rationnel grâce au théorème de Parikh. C'est-à-dire qu'on considère que les arbres sont non ordonnés : l'ordre des successeurs d'un noeud n'est pas significatif. Ce qui nous amène à nous placer dans le monoïde commutatif libre dont les parties rationnelles sont des ensembles semi-linéaires et on se repose alors sur des résultats classiques de l'arithmétique de Presburger. Nous proposons un algorithme de vérification de l'opacité du système. On montre par la suite que la complexité en temps de cet algorithme est exponentielle-complète. Des techniques de corrections de l'opacité sont aussi proposées dans ce chapitre. La première est la technique de contrôle, étudiée préalablement dans le cadre des systèmes à événements discrets [BBB+07]. La grammaire obtenue par contrôle reconnait les mêmes arbres que la grammaire de départ à l'exception des arbres pour lesquels l'opacité peut être mis en défaut. La deuxième technique consiste au contraire à étendre le langage de la grammaire de départ : on introduit du "bruit" (d'autres documents jugés conformes) afin de rétablir l'opacité. Pour cela on cherche à étendre la grammaire de façon minimale en s'inspirant des techniques d'extensions conservatives de schéma XML [BDA+04b]. On terminera ce chapitre par un problème ouvert.

Ces trois premiers chapitres constituent une version étendue de l'article [BD14].

Nous avons juste rajouté quelques exemples et le résultat concernant la complexité de la vérification de l'opacité. Une version préalable de cet article a été présenté au CARI [BD12].

Le chapitre 4 quant à lui traite de l'implémentation dans le langage fonctionel Haskell des différents algorithmes proposés dans la thèse. On commençe par une présentation de l'outil *TaPAS* pour *Talence Presburger Arithmetic Suite* qui permet de calculer l'ensemble des solutions associées à une formule de Presburger. Cet outil permet de générer l'ensemble des arbres modulo commutations. Le lien entre le code de TaPAS, écrit dans le langage C et notre code Haskell est réalisé grâce à l' interface FFI (Foreign Function Interface) d'Haskell. Ensuite nous présentons une implémentation en Haskell des algorithmes testant la conformité d'un arbre modulo commutation à une grammaire d'arbres, la projection d'un tel arbre, la grammaire quotient obtenue par projection et enfin la vérification de l'opacité.

Le document est complété par une annexe dans laquelle on rappelle quelques résultats sur l'arithmétique de Presburger. On y détaille les passages entre ensemble semi-linéaires, formules de Presburger et automates.

Chapitre 1

Conformité d'un artefact

Les arbres construits sur un alphabet Ω —c'est-à-dire un ensemble fini— sont définis inductivement comme suit

$$t \ := \omega(t_1, \ldots, t_n) \qquad \omega \in \Omega$$

ω est l'étiquette attachée à la racine de l'arbre et $t_1, \ldots, t_n$ est la liste des sous arbres se trouvant sous le noeud racine. Cette liste peut-être vide ($n = 0$) auquel cas l'arbre $t = \omega(\,)$ est réduit à son étiquette —on pourra écrire de façon abrégée $t = \omega$ dans ce cas—. Une forêt est une liste d'arbres.

Un artefact est un ensemble de documents constituant le dossier pour un client d'un certain service. Ces documents ont une structure d'arbre. Les informations attachées aux noeuds d'un arbre sont données par des paires attributs/valeurs qui permettent de collecter les informations pertinentes utiles pour le traitement du dossier. Néanmoins comme on aura besoin de travailler avec un ensemble fini d'étiquettes on suppose donnée une abstraction finie de ces informations sous la forme d'un alphabet Ω utilisé pour étiqueter les différents noeuds des documents. La structure arborescente reflète l'organisation logique du document : $d = \omega(d_1, \ldots, d_n)$ signifie que d est un document portant une information ω et dont $d_1, \ldots, d_n$ sont les constituants (les sous dossiers). Comme une forêt $f = d_1 \ldots d_n$ peut-être encodée par l'expression $\sharp(d_1, \ldots, d_n)$ et quitte à rajouter le symbole $\sharp$ à l'ensemble Ω, nous pourrons supposer qu'un artefact est également donné par un arbre.

Dans ce chapitre nous nous intéressons à caractériser la conformité d'un arbre selon une grammaire (d'arbres). Cela permet de fixer la structure logique du document, c'est-à-dire la façon dont ses différents constituants sont organisés. La notion de grammaire considérée ici correspond aux automates d'arbres (et de forêts) introduits par Murata et al [BKMW01]. Il s'agit d'une définition assez générale dans laquelle l'arité d'un noeud n'est pas fixe —comme cela est le cas pour une grammaire ordinaire, c'est-à-dire pour une signature multi sortes—.

Ceci est utile lorsque parmi des constituants d'un sous dossier on peut trouver une liste d'éléments d'une certaine catégorie (par exemple la liste des ouvrages d'une bibliothèque) et dont le nombre n'est par conséquent pas déterminé, ou lorsque certains éléments sont optionnels. Le plus simple est alors d'utiliser une expression régulière pour décrire ce qu'on est censé y trouver par exemple la règle

$$A \;\rightarrow\; \omega_1\langle B^*(A+\varepsilon)\rangle + \omega_2\langle\varepsilon\rangle$$

s'interprète de la manière suivante. A et B sont des sortes (aussi appelées catégories syntaxiques). L'ensemble des sortes est fini et sert à cataloguer les différents constituants d'un dossier en un nombre (donc fini) de catégories, par exemple ceci est un RIB, cet autre élément est une adresse ou un dossier de réservation d'hôtel. Comme dit plus haut les étiquettes $\omega \in \Omega$ sont des informations extraites des données se trouvant en racine du dossier (afin de travailler ici aussi avec un ensemble fini de possibilités). La règle précédente peut se lire alors comme suit. Un arbre est bien formé et est de la sorte A si l'information extraite à sa racine est ω_1 et il possède comme fils une suite d'éléments de sorte B —dont le nombre n'est pas fixé— et possiblement un élément de sorte A, ou bien l'information extraite à sa racine est ω_2 et dans ce cas il ne doit avoir aucun successeur.

Avant de définir formellement les grammaires nous introduisons la notion plus générale d'algèbre qui est mathématiquement plus simple et donc plus adaptée pour énoncer et prouver des résultats techniques. Une algèbre est un mécanisme permettant d'extraire (ou de synthétiser) des informations à partir du document par induction sur sa structure. Exécuter une requête sur le document pourra ainsi se faire en évaluant le document selon une algèbre. Si l'information ainsi extraite est une valeur de vérité (la valeur retournée par l'évaluation est booléenne) alors l'algèbre peut être vue comme une propriété des documents. En particulier un secret, vue comme une propriété des artefacts, sera identifié à une algèbre particulière.

Définition 1.1. *Une Ω-algèbre de support D est une application $\mathcal{A} : \Omega \times D^* \rightarrow D$. On note $\omega^{\mathcal{A}} : D^* \rightarrow D$, appelée **fonction d'interprétation** de $\omega \in \Omega$, l'application définie par : $\omega^{\mathcal{A}}(x_1, \ldots, x_n) = \mathcal{A}(\omega, x_1, \ldots, x_n)$. L'ensemble $T(\Omega)$ des arbres étiquetés sur Ω est le plus petit ensemble tel que*

$$(t_1, \ldots t_n \in T(\Omega) \;\;\wedge\;\; \omega \in \Omega) \implies \omega(t_1, \ldots, t_n) \in T(\Omega)$$

La valeur $t^{\mathcal{A}}$ d'un arbre $t \in T(\Omega)$ selon l'algèbre $\mathcal{A}$ est définie inductivement par :

$$t = \omega(t_1, \ldots, t_n) \implies t^{\mathcal{A}} = \omega^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$$

Si $\mathcal{A}$ est une algèbre et $x \in D$ un élément de son support on pose

$$L(\mathcal{A}, x) = \left\{ t \in T(\Omega) \;\middle|\; t^{\mathcal{A}} = x \right\}$$

Notations. — Notons $\mathcal{A}_{\omega,x} = \left(\omega^{\mathcal{A}}\right)^{-1} = \{u \in D^* \mid \mathcal{A}(\omega,u) = x\}$ pour $\omega \in \Omega$ et $x \in D$. Pour un symbole ω donné ces ensembles sont donc disjoints : $x \neq y \implies \mathcal{A}_{\omega,x} \cap \mathcal{A}_{\omega,y} = \emptyset$. On note $\mathcal{A}_\omega = \bigcup_{x \in D} \mathcal{A}_{\omega,x}$ leur union. De façon similaire nous associons à $x \in D$ l'ensemble $\mathcal{A}_x = \bigcup_{\omega \in \Omega} \mathcal{A}_{\omega,x}$.

Une algèbre est non déterministe si le résultat de l'évaluation peut produire différentes valeurs. C'est-à-dire qu'il s'agit d'une fonction $\mathcal{A} : \Omega \times D^* \to \wp(D)$. On peut alors de façon classique se ramener au cas déterministe en passant aux parties c'est-à-dire en remplaçant le support D de l'algèbre par l'ensemble $\wp(D)$ des parties de D.

Remarque. — **Algèbre non-déterministe** — Une application $\mathcal{A} : \Omega \times D^* \to \wp(D)$, appelée algèbre non-déterministe, est assimilée à l'algèbre $\mathcal{A}^d : \Omega \times \wp(D)^* \to \wp(D)$ définie sur $\wp(D)$ par

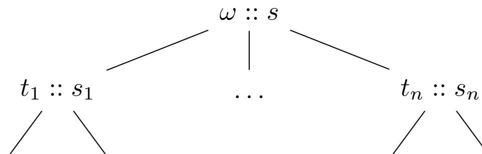
$$\mathcal{A}^d(\omega, X_1 \cdots X_n) = \bigcup \{\mathcal{A}(\omega, x_1 \cdots x_n) \mid \forall i \in \{1, \ldots n\} \;\; x_i \in X_i\}$$

Le langage d'une algèbre non déterministe est ainsi donné par $L(\mathcal{A}, x) = L(\mathcal{A}^d, x)$, c'est-à-dire $L(\mathcal{A}, x) = \left\{t \in T(\Omega) \;\middle|\; x \in t^{\mathcal{A}}\right\}$.

La première propriété d'un artefact qu'il convient de vérifier est sa conformité à un certain schéma donné par une grammaire. Comme mentionné plus haut la notion de grammaire utilisée ici s'inspire des automates d'arbres de [BKMW01].

Définition 1.2 (Grammaire)**.** *Une grammaire $G = (\Omega, \Xi, \mathcal{L})$ sur Ω est la donnée d'un ensemble fini Ξ de* ***sortes*** *et d'un langage régulier $\mathcal{L}_{\omega,s} \subseteq \Xi^*$ associé à chaque $\omega \in \Omega$ et $s \in \Xi$. On note $G \vdash t :: s$ pour signifier que t est un arbre conforme à la grammaire G et est de sorte s. On note $L(G, s) = \{t \mid G \vdash t :: s\}$ cet ensemble. Ces ensembles sont définis inductivement comme suit :*

$$(\forall i \in \{1, \ldots, n\} \;\; G \vdash t_i :: s_i \;\; \wedge \;\; s_1 \cdots s_n \in \mathcal{L}_{\omega,s}) \;\; \Rightarrow \;\; G \vdash \omega(t_1, \ldots, t_n) :: s$$



$$(\omega, s) \longmapsto \mathcal{L}_{\omega,s} \in Rat(\Xi^*)$$

que l'on note $L(G) = \bigcup_{s \in \Xi} L(G,s)$ où $L(G,s) = \{t \mid G \vdash t :: s\}$. L'ensemble $L(G)$ est un langage d'arbre, définie comme une partie de $T(\Omega)$.

Le langage régulier $\mathcal{L}_{\omega,s} \subseteq \Xi^*$ décrit les sortes possibles des constituants (noeuds fils) d'un noeud de sorte s. Par exemple $\mathcal{L}_{\omega,A} = B^* + C$ exprime le fait qu'un noeud ω sera de sorte A si ses successeurs immédiats sont soit une liste (éventuellement vide) de noeuds de sorte B soit un unique noeud de sorte C. Remarquons que ces noeuds ont une arité variable : l'étiquette d'un noeud ne caractérise pas le nombre de ses successeurs.

Notations. — **Grammaire sous forme de productions** — Une grammaire sera présentée syntaxiquement sous la forme d'un ensemble de **productions** $A \to \omega\langle E\rangle$ où E est une expression régulière telle que $L(E) = \mathcal{L}_{\omega,A}$ lorsque cet ensemble est non vide.

Exemple 1.3. *Un exemple de grammaire $G = (\Omega, \Xi, \mathcal{L})$ présentée sous forme de productions avec $\Xi = \{A, B, C, D\}$ et $\Omega = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7, \omega_8\}$:*

$$\mathcal{L} = \begin{cases} A & \to & \omega_1\langle BC\rangle + \omega_2\langle\varepsilon\rangle \\ B & \to & \omega_3\langle BD\rangle + \omega_4\langle\varepsilon\rangle \\ C & \to & \omega_5\langle CA\rangle + \omega_6\langle CCB\rangle + \omega_7\langle D\rangle \\ D & \to & \omega_8\langle AB\rangle \end{cases}$$

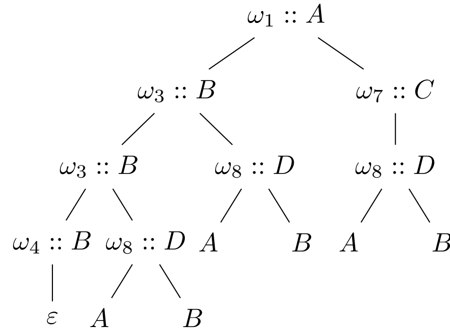


Fig. 1.1 – Un arbre conforme à la grammaire de l'exemple 1.3

La vérification de la conformité d'un arbre peut-être implémentée par un automate à pile. Cet automate à pile sera déterministe si $s \neq s' \Rightarrow \mathcal{L}_{\omega,s} \cap \mathcal{L}_{\omega,s'} = \emptyset$, c'est-à-dire que la sorte d'un noeud est caractérisée par son étiquette et la sorte de ses successeurs. Lorsque cette propriété est vérifiée la grammaire est dite

déterministe. Comme dans le cas des algèbres, une grammaire non-déterministe peut être transformée en une grammaire déterministe équivalente (c'est-à-dire ayant le même langage).

Définition 1.4 (Déterminisation d'une grammaire)**.** *La déterminisation d'une grammaire $G = (\Omega, \Xi, \mathcal{L})$ est la grammaire déterministe $G^d = (\Omega, \wp(\Xi) \setminus \{\emptyset\}, \mathcal{L}^d)$ pour laquelle $\mathcal{L}^d_{\omega,X} = \bigcup_{s \in X} \mathcal{L}_{\omega,s}$.*

La remarque suivante montre que les grammaires ne sont rien d'autres que les algèbres régulières, c'est-à-dire les algèbres dont les langages $\mathcal{A}_{\omega,x}$ sont réguliers. Par défaut néanmoins une algèbre est déterministe alors qu'une grammaire est par défaut non-déterministe.

Remarque. — **Grammaires vs algèbres** – Une grammaire G peut-être assimilée à l'algèbre non-déterministe $\mathcal{A}$ de domaine Ξ telle que $\mathcal{A}(\omega, u) = \{s \in \Xi \mid u \in \mathcal{L}_{\omega,s}\}$ pour $\omega \in \Omega$, $s \in \Xi$ et $u \in \Xi^*$, et donc $\mathcal{A}_{\omega,s} = \mathcal{L}_{\omega,s}$. Une grammaire déterministe G peut être assimilée à l'algèbre de domaine $D = \Xi \cup \{\top\}$ où $\top$ est un symbole supplémentaire ($\top \notin \Xi$) et

$$\omega^G(s_1, \ldots, s_n) = \begin{cases} s & \textbf{si} \quad s_1 \cdots s_n \in \mathcal{L}_{\omega,s} \\ \top & \textbf{sinon} \end{cases}$$

Les opérations respectives de déterminisation des grammaires et des algèbres se correspondent, c'est-à-dire que G^d décrite dans la définition 1.4 est la déterminisation de G vue comme algèbre non-déterministe : l'élément additionnel $\top$ correspond à la partie vide qu'on avait pris soin de ne pas introduire dans la définition de G^d. La correspondance entre une grammaire déterministe et l'algèbre qui lui est associée est donnée par l'identité $L(G, s) = \{t \in T(\Omega) \mid t^G = s\}$ qui découle immédiatement des définitions 1.1 et 1.2. Un arbre est ainsi conforme à la grammaire (ou bien formé) si et seulement si $t^G \neq \top$. Remarquons que comme $\mathcal{L}_{\omega,s} \subseteq \Xi^*$ et $\top \notin \Xi$ il vient que $t = \omega(t_1, \ldots, t_n)$ est non conforme (i.e., $\omega^G(t_1^G, \ldots, t_n^G) = \top$) dès qu'un de ses sous-arbres est non conforme.

Définition 1.5 (Grammaire locale)**.** *Une grammaire est dite locale si son ensemble de sortes Ξ coïncide avec son ensemble de symboles Ω et $\mathcal{L}_{\omega,s} = \emptyset$ si $\omega \neq s$. Les productions d'une grammaire locale sont donc de la forme $A \rightarrow A\langle E \rangle$ où A est un symbole de la grammaire et E un ensemble régulier sur le même ensemble. La notation devient alors redondante, on écrira par conséquent ces productions sous la forme abrégée $A \rightarrow E$.*

Exemple 1.6. *L'arbre de la figure ci-dessous est conforme à la grammaire locale*

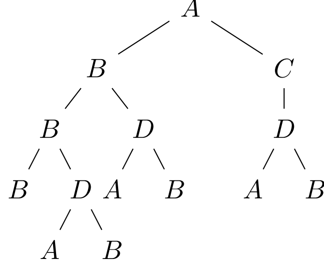


FIG. 1.2 – Un arbre conforme à une grammaire locale donnée par les règles ci-dessous.

dont les productions sont les suivantes :

$$\begin{array}{l} A \to BC + \varepsilon \\ B \to BD + \varepsilon \\ C \to CA + CCB + D \\ D \to AB \end{array}$$

NOTE. — **Langage d'une grammaire** – Dans la suite nous ferons l'hypothèse suivante sur les grammaires. L'ensemble des sortes contient un symbole spécifique $\mathbf{ax} \in \Xi$, appelé **axiome** de la grammaire, pour lequel il est associé une unique production $\mathbf{ax} \to \sharp\langle E_0\rangle$ telle que le symbole $\sharp \in \Omega$ (appelé racine) n'apparait dans aucune des autres productions de la grammaire. $t^G = s$ pour $t \in T(\Omega \setminus \{\sharp\})$ et $s \in \Xi \setminus \{\top\}$ signifie que t est bien formé de sorte s, et on pose $L(G) = \left\{t_1 \dots t_n \in T(\Omega \setminus \{\sharp\})^* \mid t_1^G, \dots, t_n^G \in L(E_0)\right\}$ le langage de la grammaire G, i.e., $t_1 \dots t_n \in L(G)$ si, et seulement si $(\sharp(t_1, \dots, t_n))^G = \mathbf{ax}$.

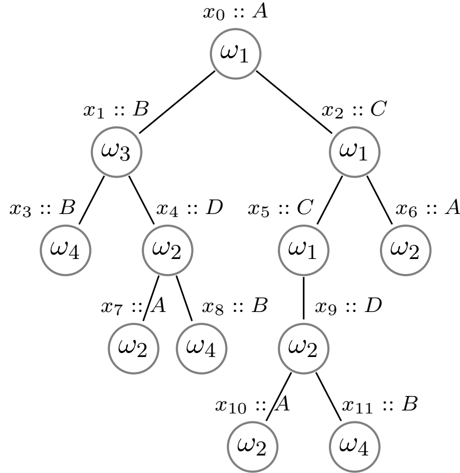
En utilisant les résultats classiques sur les langages reconnaissable d'arbres [BKMW01, CDG+08] on peut montrer que les langages des grammaires sont clos de manière effective par les opérations booléennes et qu'on peut décider de la vacuité et donc de l'inclusion de ces langages, ainsi la relation $G \leq G' \iff L(G) \subseteq L(G')$ est décidable.

Dans la remarque suivante —un peu longue mais pouvant être omise par le lecteur non intéressé— nous faisons une digression qui peut permettre au lecteur de mieux comprendre le type de propriétés d'arbres (en particulier les secrets) qu'on souhaite pouvoir exprimer en utilisant des algèbres. Cet aspect ne sera pas développé dans la suite de cet thèse.

REMARQUE. — Dans ce travail préliminaire nous ne cherchons pas à identifier une syntaxe pour exprimer les secrets sur les documents. De façon simplifiée

nous avons dit précédemment qu'on pouvait assimiler un secret (en tant que propriété des documents) à une algèbre dont le résultat retourné en racine de l'arbre est booléen (vrai ou faux). Cela n'exclut pas que le support de l'algèbre contienne d'autres valeurs de nature bien plus complexe afin de représenter des valeurs intermédiaires associées à des sous-parties du document. Dans la mesure où les informations attachées à un noeud sont des paires attributs/valeurs un formalisme particulièrement adapté pour exprimer ces calculs est celui des grammaires attribuées. Il est bien connu que l'évaluation des attributs pour une grammaire attribuée peut s'exprimer par une algèbre, néanmoins d'ordre supérieur, dont les fonctions d'interprétation permettent en chaque noeud de l'arbre de calculer les attributs synthétisés en fonction des attributs hérités. Il est montré dans [BTNJF13] que les règles sémantiques d'une grammaire attribuée peuvent se décrire par une algèbre *du premier ordre* sur une signature étendue dans laquelle aux liens d'un père vers ses fils on rajoute un lien supplémentaire de ce noeud vers son propre père afin d'accéder à son contexte. Pour des raisons d'effectivité on doit se restreindre ici à des algèbres dont le support est fini et donc considérer des abstractions des informations attachées aux noeuds et des règles sémantiques qui permettent de les calculer. Ce sera plus facile à obtenir si nous partons d'une algèbre du premier ordre. D'où l'intérêt ici de pouvoir manipuler des liens symboliques permettant d'exprimer des propriétés des documents (en particulier des secrets) qui ne seront pas nécessairement définis par induction sur leur structure physique mais pourront aussi dépendre d'informations contextuelles.

Pour ce faire, et comme illustré à la figure 1, nous pouvons décrire un arbre par un système d'équations E de la forme $x = \omega(x_1, \ldots, x_n)$ dans laquelle la variable x est dite *définie* et les variables x_i $(1 \leq i \leq n)$ sont dites *utilisées*. Les variables correspondent bijectivement aux noeuds de l'arbre. Chacune d'elles doit être définie par exactement une équation et ne doit être utilisée qu'au plus une fois. Enfin la relation de dépendance, qui est donnée par : $x \ll y$ ssi "*y est utilisée dans l'équation qui définit x*", doit être acyclique. L'évaluation de l'arbre dans une algèbre revient à associer une valeur $x^{\mathcal{A}}$ pour chaque variable de sorte que $x = \omega(x_1, \ldots, x_n) \implies x^{\mathcal{A}} = \omega^{\mathcal{A}}(x_1^{\mathcal{A}}, \ldots, x_n^{\mathcal{A}})$. Autrement dit le vecteur $\vec{x}^{\mathcal{A}}$ est solution du système $E_{\mathcal{A}}$, dit *interprétation de E dans $\mathcal{A}$*, formé des équations $x = \omega^{\mathcal{A}}(x_1, \ldots, x_n)$, c.-à-d. $\vec{x}^{\mathcal{A}} = E_{\mathcal{A}}(\vec{x}^{\mathcal{A}})$. Les composantes de ce vecteur $\vec{x}^{\mathcal{A}} : X \to D$ peuvent être calculées suivant un tri topologique du graphe de dépendance (on évalue un noeud après évaluation des noeuds dont il dépend). L'avantage de cette présentation équationnelle de l'évaluation d'un document (par opposition à une présentation plus opérationnelle à l'aide de machines à piles) est de pouvoir s'appliquer sans modification au cas où on souhaite pouvoir faire du partage de documents : l'hypothèse qu'une variable n'est jamais utilisée de façon multiple sert à caractériser les arbres mais n'a aucune utilité dans la résolution des systèmes d'équations correspondants.



$$
\begin{array}{ll}
x_0 = \omega_1(x_1, x_2) & x_6 \ = \omega_2() \\
x_1 = \omega_3(x_3, x_4) & x_7 \ = \omega_2() \\
x_2 = \omega_1(x_5, x_6) & x_8 \ = \omega_4() \\
x_3 = \omega_4() & x_9 \ = \omega_2(x_{10}, x_{11}) \\
x_4 = \omega_2(x_7, x_8) & x_{10} = \omega_2() \\
x_5 = \omega_1(x_9) & x_{11} = \omega_4()
\end{array}
$$

$$
\begin{array}{l}
A \quad \textbf{where} \\
\ A \rightarrow \omega_1\langle BC\rangle \ + \ \omega_2\langle\varepsilon\rangle \\
\ B \rightarrow \omega_3\langle BD\rangle \ + \ \omega_4\langle\varepsilon\rangle \\
\ C \rightarrow \omega_1\langle CA\rangle \ + \ \omega_4\langle CCB\rangle \ + \ \omega_1\langle D\rangle \\
\ D \rightarrow \omega_2\langle AB\rangle
\end{array}
$$

FIG. 1.3 – Un arbre vu comme un système d'équations dont la résolution dans une algèbre (par exemple une grammaire déterministe) donne la valeur de l'arbre. Le préambule "L **where** ..." à la liste des productions de la grammaire représente la production $\mathbf{ax} \rightarrow \sharp\langle L\rangle$ associée à son axiome (qui n'est, ainsi que le symbole en racine $\sharp$, pas représenté explicitement).

On peut également lever l'hypothèse que le graphe de dépendance est acyclique en résolvant le système d'équation $\vec{x}^{\mathcal{A}} = E_{\mathcal{A}}(\vec{x}^{\mathcal{A}})$ de manière itérative. Il faut pour cela supposer que le domaine de l'algèbre est un treillis complet (dont le plus petit élément $\perp$, la valeur indéfinie, est la valeur partagée par tout objet et le plus grand élément $\top$, la valeur contradictoire, est une valeur qui ne peut-être prise par aucun objet) et que les fonctions d'interprétations sont continues (c'est-à-dire commutent aux bornes supérieurs). Dans ce cas le plus petit point fixe de l'équation $\vec{x} = E_{\mathcal{A}}(\vec{x})$, noté $E^{\dagger}_{\mathcal{A}}$ est la borne supérieure de la suite obtenue en itérant le constructeur $E_{\mathcal{A}}$ à partir du vecteur totalement indéfini $\vec{\perp}$ (dont toutes les composantes sont égales à $\perp$), c'est-à-dire que $E^{\dagger}_{\mathcal{A}} \bigvee E^{n}_{\mathcal{A}}(\vec{\perp})$. Certaines hypothèses sur le support de l'algèbre (comme le fait qu'il soit de hauteur bornée) garantissent que la suite croissante $\langle E^{n}_{\mathcal{A}}(\vec{\perp})\rangle_{n\in\mathbb{N}}$ est stationaire, c'est-à-dire se stabilise toujours à partir d'un rang m pour lequel $E^{\dagger}_{\mathcal{A}} = E^{m}_{\mathcal{A}}(\vec{\perp})$.

Un cycle dans la relation de dépendance semble dire qu'un document est une composante de lui-même, ce qui ne fait pas de sens. Néanmoins dans certains cas la relation de dépendance ne correspondra pas uniquement à la relation de contenance (la structure physique sous-jacente du document) mais aussi à des liens symboliques qui peuvent être utilisés pour rechercher des informations. C'est-à-dire qu'on trouvera un chemin dans le graphe de dépendance de x vers y si le calcul

d'un certain attribut du noeud x peut dépendre de la valeur de certains attributs du noeud y. Même si on ne doit pas trouver de cycle dans les dépendances entre les attributs on pourra en trouver entre les noeuds qui portent ces attributs dès que la valeur d'un attribut peut dépendre d'informations provenant du contexte de ce noeud (et non seulement de ses sous-arbres).

Remarque. — **Grammaires vs signatures multi-sortes** – Les grammaires sont une généralisation des signatures multisortes qui peuvent-être assimilées aux grammaires dans lesquelles pour chaque symbole $\omega \in \Omega$ les langages $\mathcal{L}_{\omega,s}$ sont tous vides sauf un qui est réduit à un seul mot. C'est-à-dire que chaque symbole a une arité et une sorte qui sont fixées : on note $\omega :: s_1 \dots s_n \to s$ lorsque $\mathcal{L}_{\omega,s} = \{s_1 \cdots s_n\}$ (et donc $s \neq s' \implies \mathcal{L}_{\omega,s'} = \emptyset$). Inversement, à toute grammaire déterministe on peut associer une signature multisorte dont les opérateurs sont les paires $\langle\omega, u\rangle$ pour $u = s_1 \dots s_n \in \Xi^*$ telles que $G(\omega, u) = s \neq \top$ il s'agit alors d'un opérateur d'arité u et de sorte s, c'est-à-dire $\langle\omega, u\rangle :: s_1 \dots s_n \to s$. Rappelons qu'une algèbre $\mathcal{A}$ pour une signature multi-sorte est donnée par un ensemble non vide $\mathcal{A}_s$ associé à chaque sorte $s \in \Xi$ et une fonction d'interprétation $\omega^{\mathcal{A}} : \mathcal{A}_{s_1} \dots \mathcal{A}_{s_n} \to \mathcal{A}_s$ associée à chaque opérateur $\omega :: s_1 \dots s_n \to s$. Les algèbres pour la signature multisorte associée à une grammaire déterministe G sont les Ω-algèbres $\mathcal{A} : \Omega \times D^* \to D$ pour lesquelles il existe une application surjective $\varphi : D \twoheadrightarrow \Xi$ telle que $\mathcal{L}_{\omega,\varphi(x)} = \{\varphi(x_1) \dots \varphi(x_n) \mid \mathcal{A}(\omega, x_1 \cdots x_n) = x\}$. Autrement dit $G \circ \langle id_\Omega, \varphi^*\rangle = \varphi \circ \mathcal{A}$. Nous dirons par analogie que ces algèbres sont les **algèbres de la grammaire**.

Un artefact (liste de documents) est conforme s'il appartient au langage de la grammaire et en particulier chacun des documents qui le constitue est bien formé. Une fois que la conformité de l'artefact est acquise on peut en extraire de l'information en l'évaluant selon des algèbres de la grammaire au sens de la remarque précédente.

Chapitre 2

Abstraction d'un artefact

Dans la section précédente nous nous sommes intéressés à la structure logique d'un artefact (donnée par une grammaire) et à son évaluation selon une algèbre. Nous allons maintenant nous intéresser aux données qu'il contient. Les informations attachées à un noeud correspondent à son étiquette $\omega \in \Omega$. Nous pouvons voir un artefact comme des boîtes imbriquées avec pour chacune d'entre elles les informations qui lui sont attachées. Une abstraction (ou observation) consiste à supprimer toutes les boîtes associées à des sortes "non visibles" en supprimant par la même occasion les données qui leur correspondent. L'effet de cette transformation sur la structure du document peut se décrire comme suit.

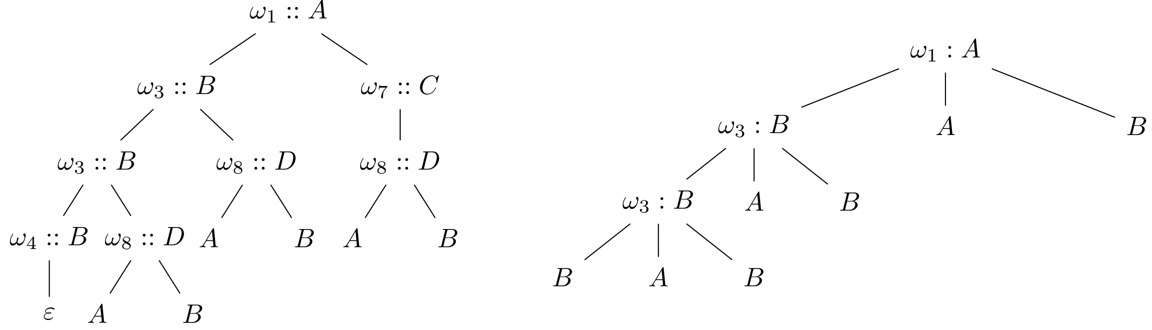
Définition 2.1. *La fonction de projection $p_{\Xi'} : T(\Omega) \rightarrow T(\Omega)^*$ associée à un sous-ensemble $\Xi' \subseteq \Xi$ (les sortes visibles) est donnée pour $t = \omega(t_1, \ldots, t_n)$ par :*

$$p_{\Xi'}(t) = \begin{cases} \omega(p_{\Xi'}(t_1) \ldots p_{\Xi'}(t_n)) & si \quad t^G \in \Xi' \\ p_{\Xi'}(t_1) \cdots p_{\Xi'}(t_n) & si \quad t^G \notin \Xi' \end{cases}$$

Les sortes visibles contiennent toujours le symbole en racine : $\sharp \in \Xi'$.

Ainsi si la sorte d'un arbre est visible sa projection est un arbre sinon il s'agit d'une suite d'arbres —i.e., une forêt—.

Exemple 2.2. *L'arbre de la figure 1.3 et son projeté suivant le sous-ensemble de sortes visibles $\Xi' = \{A, B\}$:*



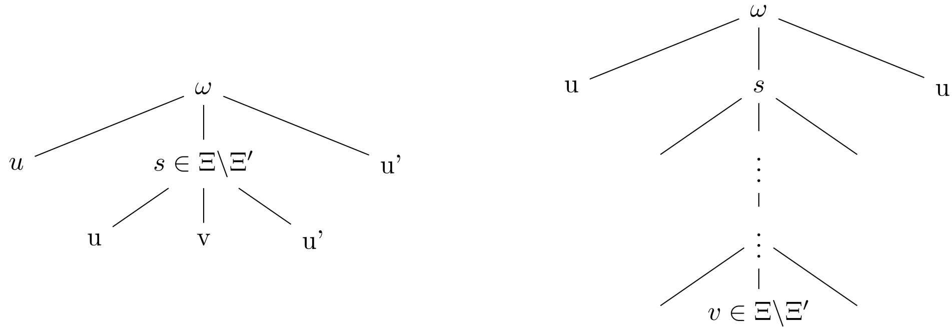
Définition 2.3. *Si $\pi = p_{\Xi'}$ est la projection associée à $\Xi' \subseteq \Xi$, on pose $\mathcal{A} = G/\pi$ l'algèbre non-déterministe $\mathcal{A} : \Omega \times \Xi'^* \to \wp(\Xi')$ définie par*

$$\mathcal{A}(\omega, v) = \{s \in \Xi' \mid \exists u \in \mathcal{L}_{\omega,s} \quad u \vdash^* v\}$$

où $\vdash^*$ est la fermeture réflexive et transitive de la relation

$$\vdash\ = \{(u \cdot s \cdot u', u \cdot v \cdot u' \mid s \in \Xi \setminus \Xi' \text{ et } v \in \mathcal{L}_s\}$$

Ainsi $\quad \mathcal{A}^d(\omega, X_1 \cdots X_n) = \{s \in \Xi' \mid \exists u \in \mathcal{L}_{\omega,s}\ \exists x_i \in X_i \quad u \vdash^* x_1 \cdots x_n\}.$



Remarque. — **Dérivation maximale** – On note $u \vdash^*_{\max} v$ lorsque $u \vdash^* v$ et cette dérivation est maximale, c'est-à-dire qu'il n'existe pas de w dans lequel v se dérive. Si les langages $\mathcal{L}_s = \cup_\omega \mathcal{L}_{\omega,s}$ sont tous non vides —hypothèse que nous ferons par la suite— alors

$$u \vdash^*_{\max} v \iff u \vdash^* v \text{ et } v \in (\Xi')^*$$

Le théorème suivant est le principal résultat de cet thèse. Il montre la correction de la grammaire quotient en ce sens qu'elle reconnait exactement les projections des arbres conformes à la grammaire de départ.

Théorème 2.4. *$L(G/\pi) = \pi(L(G))$.*

Afin d'établir ce théorème nous introduisons sur les arbres une relation, indexée par les contextes, qui correspond à la relation de dérivation $\vdash$ sur les sortes.

Définition 2.5 (Contextes)**.** *Les opérateurs dérivés, ou* ***contextes****, $C \in \Omega^{@}$ sont les arbres construits sur la signature $\Omega \cup \{[\,]\}$ obtenue à partir de Ω en ajoutant un symbole spécial $[\,]$ figurant la position d'un argument du contexte. Les contextes sont ainsi définis par la syntaxe BNF suivante :*

$$C \;:=\; [\,] \mid \omega(C_1, \ldots, C_n)$$

*L'****arité*** *d'un contexte, correspondant à son nombre d'arguments, est défini inductivement par $\mathbf{ar}([\,]) = 1$ et $\mathbf{ar}(\omega(C_1, \ldots, C_n)) = \mathbf{ar}(C_1) + \cdots + \mathbf{ar}(C_n)$ (avec par convention $\mathbf{ar}(\omega()) = 0$ correspondant au cas où $n = 0$). On note $\Omega_n^{@}$ l'ensemble des contextes d'arité n et $\omega_n \in \Omega_n^{@}$ le contexte d'arité n associé à l'opérateur ω, il s'agit du contexte $\omega_n = \omega([\,], \ldots, [\,])$ contenant exactement n arguments. Un contexte $C \in \Omega_n^{@}$ d'arité n s'interprète selon une algèbre $\mathcal{A} : \Omega \times D^* \to D$ comme l'opérateur $C^{\mathcal{A}} : D^n \to D$ donné par :*

(1) $[\,]^{\mathcal{A}}(x) = x$

(2) $(\omega(C_1, \ldots, C_n))^{\mathcal{A}}(v) = \omega^{\mathcal{A}}(C_1^{\mathcal{A}}(v_1), \ldots, C_n^{\mathcal{A}}(v_n))$
où $|v_i| = \mathbf{ar}(C_i)$ *et* $v_1 \cdots v_n = v$.

En particulier on assimilera un contexte avec son interprétation dans l'algèbre libre $F : \Omega \times T(\Omega)^* \to T(\Omega)$ (pour laquelle $\omega^F(t_1, \ldots, t_n) = \omega(t_1, \ldots, t_n)$), c'est-à-dire que l'expression $C(t_1, \ldots, t_n)$ sera définie si, et seulement si, ω est d'arité n, et dans ce cas elle correspond à l'arbre obtenu en substituant les arbres $t_1, \ldots, t_n$ aux arguments du contexte pris dans leur ordre d'apparition. L'écriture $t = C(t_1, \ldots, t_n)$ correspond à une ***décomposition*** *d'un arbre en un contexte et une suite d'arbres associés aux arguments du contexte.*

Nous allons décomposer un arbre *de sorte visible* $t \in T(\Omega)_{\Xi'} = \{t \in T(\Omega) \mid t^G \in \Xi'\}$ sous la forme $t = C(t_1, \ldots, t_n)$ où C est le plus petit contexte non trivial (i.e., $C \neq [\,]$) pour lequel $t_i^G \in \Xi'$. En particulier si tous les sous arbres stricts t' de t sont tels que $t' \notin \Xi'$, on aura $t = C$ (et donc le contexte ne contiendra aucun argument). Cette décomposition est donnée par la relation $t \gg_C t_1 \ldots t_n$ définie ci-dessous.

Définition 2.6 (Décomposition d'un arbre)**.** *La relation $\vdash \subseteq T(\Omega) \times \Omega^{@} \times T(\Omega)^*$, où $(t, C, t_1 \ldots, t_n) \in \vdash$ est noté $t \vdash_C t_1 \ldots t_n$, est la plus petite relation telle que*

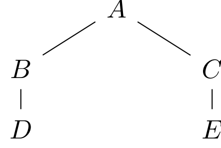
(1) $t \vdash_{[\,]} t$ *si* $t^G \in \Xi'$

(2) $t \vdash_{\omega_n} t_1 \ldots t_n$ *si* $t = \omega(t_1, \ldots, t_n)$ *et* $t^G \notin \Xi'$

(3) *Si* $t \vdash_{\omega_n} t_1 \ldots t_n$ *et* $t_i \vdash_{C_i} v_i$ *pour tout* $1 \leq i \leq n$ *alors* $t \vdash_{\omega_n(C_1, \ldots, C_n)} v$
où $v = v_1 \cdots v_n$ *est la concaténation des* v_i.

Pour $t = \omega(t_1, \ldots, t_n) \in T(\Omega)_{\Xi'}$ *on pose* $t \gg_{\omega_n(C_1,\ldots,C_n)} v$ *lorsque* $t_i \vdash_{C_i} v_i$ *où* $v_i \in (T(\Omega)_{\Xi'})^*$ *et* $v = v_1 \cdots v_n$ *est la concaténation des suites d'arbres* $v_1, \ldots, v_n$*. On pose* $t \gg v$ *si* $t \gg_C v$ *pour un certain contexte* C*.*

Exemple 2.7. *On considére l'arbre t suivant :*



La décomposition de l'arbre est donnée, en appliquant (1),(2) et (3) de la définition ci-dessus par :

(2) $\Rightarrow\ t \vdash_A B(D), C(E)$ *ensuite*
(2) $\Rightarrow\ B(D) \vdash_B D$
(1) $\Rightarrow\ D \vdash_{[]} D$
(3) $\Rightarrow\ B(D) \vdash_B D$
$\vdots$
(3) $\Rightarrow\ C(E) \vdash_C E$
(3) $\Rightarrow\ t \vdash_{A(B,C)} DE$
Alors $t >>_a de$

La correspondance entre cette décomposition d'un arbre et la relation de dérivation sur les sortes est donnée par les lemmes suivants.

Lemme 2.8. $t \vdash_C t_1 \cdots t_n \implies t^G \vdash^* t_1^G \cdots t_n^G$

Démonstration. La démonstration s'effectue par récurrence sur la preuve de la relation $t \vdash_C t_1 \cdots t_n$. Le cas de base se décline suivant la valeur de t^G :

$t^G \in \Xi'$**.** On a d'une part $t \vdash_{[]} t$ et d'autre part $t^G \vdash^* t^G$ par reflexivité de cette relation.

$t^G \notin \Xi'$**.** Si t est de la forme $t = \omega(t_1, \ldots, t_n)$ on déduit que $t \vdash_{\omega_n} t_1 \ldots t_n$. D'autre part $t_1^G \ldots t_n^G \in \mathcal{L}_{\omega, t^G}$ et donc $t^G \vdash t_1^G \ldots t_n^G$.

Pour le cas général supposons $t = \omega(t_1, \ldots, t_n)$ et $t_i \vdash_{C_i} v_i$ pour tout $1 \leq i \leq n$. On a dans ce cas $t \vdash_{\omega_n(C_1,\ldots,C_n)} v_1 \ldots v_n$. Par hypothèse de récurrence $t_i^G \vdash^* v_i^G$ en convenant que $(t'_1 \ldots t'_k)^G = (t'_1)^G \ldots (t'_k)^G$. Par ailleurs de $t_1^G \ldots t_n^G \in \mathcal{L}_{\omega, t^G}$ il vient $t^G \vdash t_1^G \ldots t_n^G$, et donc $t^G \vdash^* v_1^G \ldots v_n^G$. □

Le lemme suivant en est une réciproque partielle où on se restreint aux dérivations maximales, c'est-à-dire que les sortes des arbres $t_1, \ldots, t_n$ sont visibles (la sorte s par contre ne le sera généralement pas).

Lemme 2.9. $s \vdash^*_{\max} t_1^G \cdots t_n^G \implies \exists t \in T(\Omega)\ \exists C \in \Omega_n^@ \quad t \vdash_C t_1 \cdots t_n \ \ et\ \ t^G = s$

Démonstration. La preuve est par récurrence sur la longueur de la dérivation de $\vdash^*$.

Dérivations de longueur nulle : ceci correspond au cas où $s \in \Xi'$ —puisque la dérivation est maximale— avec $n = 1$ et $s = t_1^G$. Le résultat découle de $t_1 \vdash_{[]} t_1$.

Cas général : la dérivation $s \vdash^* t_1^G \cdots t_k^G$ peut se décomposer en $s \vdash s_1 \ldots s_n$ —ce qui signifie que $s_1 \ldots s_n \in \mathcal{L}_{\omega,s}$ pour un certain opérateur $\omega \in \Omega$— et, pour tout $1 \le i \le n$, $s_i \vdash v_i$ tels que $v_1 \cdots v_n = t_1^G \cdots t_k^G$. Puisque les t_i^G sont visibles ces dérivations sont maximales. Par hypothèse de récurrence il existe un arbre $t_i \in T(\Omega)$ et un contexte $C_i \in \Omega^@$ tels que $t_i^G = s_i$ et $t_i \vdash_C v_i$. L'arbre $t = \omega(t_1, \ldots, v_n)$ vérifie $t^G = s$ et $t \vdash_{\omega n(C_1,\ldots,C_n)} t_1 \ldots t_n$.

□

Lemme 2.10. $\big(\exists s_1 \ldots s_n \in \mathcal{L}_{\omega,s} \ \ t.q.\ \ s \in \Xi' \ \ et\ \ s_1 \ldots s_n \vdash^*_{\max} (t'_1)^G \ldots (t'_k)^G\big) \implies$

$\big(\exists t \in T(\Omega) \ \ t.q.\ \ t \gg t'_1 \ldots t'_k \ \ et\ \ t^G = s\big)$

Démonstration. Supposons $s_1 \ldots s_n \vdash^*_{\max} (t'_1)^G \ldots (t'_k)^G$ pour $s \in \Xi'$ et $s_1 \ldots s_n \in \mathcal{L}_{\omega,s}$. Par le lemme 2.9 on déduit l'existence d'arbres t_i et de contextes C_i tels que $t_i \vdash_{C_i} v_i$, $v_1 \ldots v_n = t'_1 \ldots t'_k$ et $t_i^G = s_i$. Posons $t = \omega(t_1, \ldots, t_n)$. De $t_i \vdash_{C_i} v_i$, $v_1 \ldots v_n = t'_1 \ldots t'_k$ on déduit $t \gg t'_1 \ldots t'_k$. Par ailleurs de $s_1 \ldots s_n \in \mathcal{L}_{\omega,s}$ et $t_i^G = s_i$ on déduit $t^G = s$. □

Tout arbre $t \in T(\Omega)_{\Xi'}$ dont la sorte est visible peut se décomposer sous la forme $t = C(t_1, \ldots, t_n)$ dans lequel C est un contexte non trivial (i.e., $C \neq [\,]$) et $t_1, \ldots, t_n$ sont des arbres de sortes visibles. La relation $t \gg_C t_1 \cdots t_n$ correspond à une telle décomposition, plus précisémment c'est celle pour laquelle le contexte C est minimal. Le contexte C et les arbres $t_1, \ldots, t_n$ tels que $t \gg_C t_1 \cdots t_n$ sont donc déterminés de façon unique à partir de t. Autrement dit la relation $\gg$ détermine une application $\gg: T(\Omega)_{\Xi'} \to (T(\Omega)_{\Xi'})^*$. Lorsque $t \gg t_1 \cdots t_n$ les arbres $t_1, \ldots, t_n$ sont strictement plus petits que t —où la taille d'un arbre est donnée par son nombre de noeuds— et donc si $P \subseteq T(\Omega)_{\Xi'}$, vue comme une propriété des arbres dont la sorte est visible, vérifie les conditions suivantes [1]

1. $t \gg \varepsilon \implies P(t)$

[1] On peut voir le premier cas comme un particulier du second si on admet le cas $n = 0$ en convenant que la liste $t_1 \cdots t_n$ représente alors la liste vide. La même convention s'applique pour la définition inductive des arbres. Avec cette convention nous évitons de considérer le cas de base dans la récurrence sur la décomposition par $\gg$ ainsi que dans la récurrence sur la structure des arbres puisque à chaque fois les cas de bases apparaissent comme cas particuliers du cas général.

2. $t \gg t_1 \cdots t_n \ \wedge \ P(t_1) \ \wedge \cdots P(t_n) \implies P(t)$

alors $P = T(\Omega)_{\Xi'}$. Ceci nous donne un principe d'induction qui nous permet de raisonner par récurrence sur l'ensemble des arbres dont la sorte est visible en utilisant la décomposition associée à la relation $\gg$. On note par ailleurs que

$$t \gg_\omega t'_1 \dots t'_k \implies \pi(t) = \omega(\pi(t'_1) \cdots \pi(t'_k))$$

Nous disposons maintenant de tous les ingrédients nous permettant d'établir le théorème 2.4.

Démonstration. **[du théorème 2.4]**
Nous montrons que $L(G/\pi, s) = \pi(L(G, s))$ pour tout $s \in \Xi'$. C'est-à-dire que les deux ensembles suivants coïncident $\{\pi(t) \mid t^G = s\} = \{t \mid s \in t^{G/\pi}\}$. Pour cela nous montrons successivement que *(i)* $t^G \in (\pi(t))^{G/\pi}$ lorsque $t^G \in \Xi'$ et *(ii)* $s \in t'^{G/\pi} \implies (\exists t \text{ t.q. } t^G = s \text{ et } \pi(t) = t')$.

(i) On raisonne par récurrence sur la décomposition de t donnée par la relation $\gg$. Supposons $t \gg t'_1 \dots t'_k$. Cela signifie que $t = \omega(t_1, \dots, t_n)$ tel que $t_1^G \cdots t_n^G \in \mathcal{L}_{\omega, t^G}$, $t_i \vdash_{C_i} v_i$ avec $v_i \in (T(\Omega)_{\Xi'})^*$ pour $1 \leq i \leq n$ et $v_1 \cdots v_n = t'_1 \cdots t'_k$. Par le lemme 2.8 il vient que $t_i^G \vdash^* v_i^G$ pour $1 \leq i \leq n$. Par hypothèse de récurrence on déduit que $(t'_i)^G \in \pi(t'_i)^{G/\pi}$ et donc $(t'_1)^G \cdots (t'_k)^G \in \mathcal{A}^d(\omega, t^G)$. Du fait que $\pi(t) = \omega(\pi(t'_1), \dots, \pi(t'_k))$ il vient $t^G \in \pi(t)^{G/\pi}$.

(ii) On raisonne par récurrence sur la structure de l'arbre t'. Supposons $t' = \omega(t'_1 \dots t'_k)$ et $s \in (t')^{G/\pi}$. En utilisant la définition 2.3 on en déduit l'existence de $s_1 \dots s_n \in \mathcal{L}_{\omega,s}$ et de $s'_i \in (t'_i)^{G/\pi}$ pour tout $1 \leq i \leq k$ tels que $s_1 \dots s_n \vdash^*_{\max} s'_1 \dots s'_k$. Par hypothèse de récurrence il existe des arbres t_i tels que $t_i^G = s'_i$ et $\pi(t_i) = t'_i$. De $s_i \dots s_n \in \mathcal{L}_{\omega,s}$ et $s_1 \dots s_n \vdash^*_{max} t_1^G \dots t_k^G$ on en déduit par le lemme 2.10 l'existence d'un arbre $t \in T(\Omega)_{\Xi'}$ tel que $t \gg t_1 \dots t_k$ et $t^G = s$. De $t \gg t_1 \dots t_k$ et $\pi(t_i) = t'_i$ on déduit $\pi(t) = \omega(\pi(t_1), \dots, \pi(t_k)) = \omega(t'_1 \dots t'_k) = t'$.

Comme $\sharp \in \Xi'$ le résultat découle de

$$\begin{aligned} \pi(L(G)) &= \{\pi(t_1) \dots \pi(t_n) \mid \sharp(t_1, \dots, t_n) \in L(G, \mathbf{ax})\} \\ &= \{\pi(t_1) \dots \pi(t_n) \mid \sharp(\pi(t_1), \dots, \pi(t_n)) \in L(G/\pi, \mathbf{ax})\} \\ &= L(G/\pi) \end{aligned}$$

Car $\pi : L(G, s) \to L(G/\pi, s)$ est une surjection pour tout $s \in \Xi'$. □

Chapitre 3

Opacité d'un artefact

Nous nous intéressons aux deux problèmes suivants.

PROBLÈME. — **Invocation d'un service** – Supposons qu'on veuille développer un système, manipulant des documents conformes à une grammaire $G = (\Omega, \Xi, \mathcal{L})$, en utilisant des services dont l'interface est donnée par une autre grammaire G' qui décrit les documents que le service est disposer à recevoir. Un sous ensemble $\Xi' \subseteq \Xi$ de sortes visibles peut représenter les parties du document qu'on souhaite transmettre au service pour traitement. On doit pouvoir vérifier que $p_{\Xi'}(L(G)) \subseteq L(G')$ pour s'assurer que les documents transmis par le système au service seront toujours reconnus conformes par ce dernier.

PROBLÈME. — **Opacité** – Si un ensemble reconnaissable d'arbres $S \subseteq T(\Omega)$ représente un secret vis-à-vis de l'observateur associé à la fonction d'abstraction $\phi = p_{\Xi'}$ — opérant sur les arbres conformes à une grammaire $G = (\Omega, \Xi, \mathcal{L})$ avec $\Xi' \subseteq \Xi$— alors S est opaque vis-à-vis de ϕ si et seulement si $p_{\Xi'}(S \cap L(G)) \leq p_{\Xi'}(L(G) \setminus S)$.

Le langage $\mathcal{A}_{\omega,s} = \{v \in \Xi'^* \mid \exists u \in \mathcal{L}_{\omega,s} \quad u \vdash^* v\}$ associé à l'algèbre $\mathcal{A} = (G/p_{\Xi'})$ qui caractérise les projections des documents conformes à la grammaire G, i.e., $L(\mathcal{A}) = p_{\Xi'}(L(G))$, est un *langage algébrique*. Ces langages sont rationnels si la grammaire est *non récursive*, c'est-à-dire si on ne peut trouver de cycles dans la relation de dépendance $s > s'$ correspondant au fait que s' apparait en partie droite d'une production associée à s. Dans ce cas l'algèbre $\mathcal{A}$ est une grammaire. Les grammaires sont closes de manière effective par les opérations booléennes. La complémentation et donc l'union ne préservent pas la non-circularité. Néanmoins les deux opérations qui nous intéressent, à savoir l'intersection et la différence, préservent cette propriété. Enfin on sait décider de la vacuité et donc de l'inclusion des langages de telles grammaires. Les deux problèmes ci-dessus peuvent donc être résolus de manière effective dans ce cas. En particulier on peut construire la

grammaire $\pi_{\Xi'}(S \cap G) \setminus p_{\Xi'}(G \setminus S)$ qui reconnait exactement les cas de violation du secret. Dans le cadre d'un outil interactif cette grammaire peut être utilisée pour fournir des contre-exemples —les témoins— lorsque l'opacité n'est pas assurée. Cette information peut être utilisée par le concepteur du système pour l'aider à modifier la description du workflow en vue d'en assurer l'opacité.

Il est assez naturel de considérer comme nous l'avons fait des grammaires dans lesquelles les parties droites des productions sont des langages réguliers (et non un ensemble fini de mots) car on aura souvent besoin de considérer qu'un noeud d'un artefact puisse contenir une liste de sous documents d'une certaine sorte sans que la quantité de tels documents puissent fixée *a priori*. En revanche on aura rarement besoin de considérer des grammaires récursives. Ainsi du point de vue des artefacts l'hypothèse de non circularité est tout à fait raisonnable. Néanmoins comme nous l'avons signalé plus haut dans le texte cette hypothèse restreindra la classe des secrets que nous pourront prendre en compte puisqu'on ne pourra pas exprimer des propriétés qui dépendent d'information contextuelles.

La solution alternative que nous considérons ci-dessous est d'interpréter les documents modulo permutations des sous arbres d'un noeud. Nous restons de cette façon dans le cadre rationnel grâce au thérème de Parikh qui nous dit que l'image commutative d'un langage algébrique est rationnel.

Nous considérons que l'ordre dans lequel les successeurs d'un noeud apparaissent n'est pas significatif. Pour prendre en compte cette contrainte on pourrait décider de se restreindre aux grammaires pour lesquelles les langages $\mathcal{L}_{\omega,A}$ sont clos par commutations (la plus petite congruence $\approx$ pour laquelle $\omega \cdot \omega' \approx \omega' \cdot \omega$). Mais ceci peut être beaucoup trop restrictif. Pour cette raison, nous allons plutôt considérer la relation de “conformité modulo permutations” définie comme suit.

Définition 3.1 (Conformité modulo permutations)**.** *La relation de conformité modulo permutations est la plus petite relation telle que :*

$$(\forall i \in \{1, \ldots, n\} \quad G \vdash_c t_i :: s_i \quad \wedge \quad s_1 \cdots s_n \in [\mathcal{L}_{\omega,s}]) \quad \Rightarrow \quad G \vdash_c \omega(t_1, \ldots, t_n) :: s$$

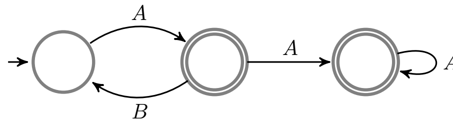
où $[L] = \{u \mid \exists v \in L \quad u \approx v\}$ est la clôture de L modulo commutations. Le ***langage modulo commutations*** *d'une grammaire est donné par :*

$$\begin{array}{ll} L_c(G,s) &= \{t \in T(\Omega) \mid G \vdash_c t :: s\} \\ L_c(G) &= \{t_1 \ldots t_n \in T(\Omega) \mid G \vdash_c \sharp(t_1, \ldots, t_n) :: \mathbf{ax}\} \end{array}$$

Remarque. — Pour que la procédure de reconnaissance soit déterministe nous supposerons, non seulement que les différents langages $\mathcal{L}_{\omega,s}$ —lorsque s varie— sont disjoints, mais que leurs clôtures commutatives le sont également : $s \neq s' \Rightarrow [\mathcal{L}_{\omega,s}] \cap [\mathcal{L}_{\omega,s'}] = \emptyset$.

Dans la mesure où la clôture par commutations d'un langage rationnel n'est généralement pas un langage rationnel, cette seconde approche est plus générale.

Exemple 3.2. *La clôture par permutation de $\mathcal{L} = (AB)^* \cdot A^*$ est l'ensemble des mots dont le nombre d'occurrences de A est supérieur ou égal au nombre d'occurrences de B. Ce dernier n'est pas rationnel (il possède un ensemble infini de résidus). On peut utiliser l'automate associé au langage régulier $\mathcal{L}$*



comme un générateur *de $[\mathcal{L}]$. Pour cela on interprète l'expression rationnelle $(AB)^*A^*$ dans N^2 en identifiant les lettres A et B avec les deux générateurs $(1,0)$ et $(0,1)$ respectivement. Au niveau de l'automate cela revient à cumuler les valeurs des étiquettes rencontrées le long d'un chemin reconnaissant de l'automate. Cet automate procure ainsi un mécanisme qui permet d'engendrer $[\mathcal{L}] = \{(x,y) \in N^2 \mid x \geq y\}$. Cet ensemble est associé à la formule $F(x,y) \equiv (\exists z) \cdot x = y + z$. Néanmoins cet automate ne peut être utilisé comme un reconnaisseur.*

Remarque. — La reconnaissance de documents modulo permutations est pris en compte dans le formalisme de *RelaxNG* à l'aide du pattern *interleave* [CM01] : il se traduit dans la syntaxe XML par le tag *interleave* qui permet de ne pas considérer l'ordre d'apparition des éléments [1]. Malheureusement, Il est diffcile de trouver les algorithmes utilisés pour la reconnaissance de ces types de documents.

Rappel. — On peut définir, de façon générale, ce qu'est une partie reconnaissable d'un monoïde M. Ce sont les images inverses $R = \varphi^{-1}(F)$ d'une partie $F \subseteq N$ d'un monoïde *fini* N par un morphisme de monoïdes $\varphi : M \to N$. Si M est finiement engendré, par les générateurs $A = \{a_1, \ldots, a_n\} \subseteq M$, la donnée (N, A, e, φ, F) où $e \in N$ est l'élément neutre de N, peut s'interpréter comme un automate dont N sont les états, $e \in N$ est l'état initial, $F \subseteq N$ sont les états finals et la fonction de transition (déterministe et totale) est donnée par $\delta(q,a) = q \cdot \varphi(a)$ pour $q \in N$ et $a \in A$. Il est facile de vérifier que $u \in R$ ssi une décomposition $v \in A^*$ de u est reconnue par cet automate, et que cela ne dépend pas du choix de la décomposition. Les parties rationnelles et reconnaissables coïncident dans le cas des monoïdes libres mais pas en général. Les parties reconnaissables d'un monoïde finiement engendré sont rationnelles. A titre d'exemple les parties reconnaissables de N^k sont les unions

[1]Xmlschemata, the interleave Pattern : books.xmlschemata.org/relaxng/relax-CHP-6-SECT-2.html

finies de "grilles rectangulaires" . Ces dernières désignent des ensembles de vecteurs de la forme $v + \sum_{i \in I} n_i \cdot e_i$ où $v \in N^k$ est un vecteur fixé, $I \subseteq \{1, \ldots, k\}$, e_i est le $i^{\text{ème}}$ vecteur unité et les $n_i \in N$ parcourent l'ensemble des entiers. Le quadrant $Q = \{(x, y) \in N^2 \mid x \geq y\}$ qui est rationnel puisque $Q = (1,0)^* \cdot (1,1)^*$ ne peut s'écrire comme une union finie de telles grilles (il s'agit d'une partie rationnelle non reconnaissable).

Les ensembles qui nous intéressent sont donc les images commutatives des langages rationnels, c'est-à-dire les parties rationnelles de N^k (où k est la taille de l'alphabet). Le fait que ces ensembles ne soient pas reconnaissables pourrait sembler constituer un obstacle pour la vérification de la conformité d'un document (modulo commutations). En fait il n'en est rien, bien que ce processus va devoir suivre une procédure un peu plus compliquée que dans le cas de base. Les parties rationnelles de N^k coïncident avec les parties semi-linéaires.

Rappel. — Une partie linéaire est l'ensemble des vecteurs de la forme $v_0 + n_1 \cdot v_1 + \cdots + n_\ell \cdot v_\ell$ (une expression affine dans laquelle les vecteurs v_0 —l'origine— et $v_1, \ldots v_\ell$ —les vecteurs de base— sont fixés et les variables n_i représentent des entiers arbitraires). Une partie semi-linéaire est une union finie de parties linéaires.

Ces ensembles semi-linéaires coïncident par ailleurs avec les ensembles définissables par des formules de Presburger, i.e., de la forme $\{(x_1, \cdots, x_n) \mid F(x_1, \cdots, x_n)\}$ où F est une formule de Presburger.

Rappel. — La logique de Presburger est le fragment de l'arithmétique dans lequel on exclut la multiplication, c'est-à-dire qu'il s'agit du calcul propositionnel avec quantification sur les entiers et avec l'addition. On peut bien sûr écrire la multiplication par une constante puisque par ex. $3x = x + x + x$. De la même manière on peut écrire des inéquations : par ex. $x \leq y$ est une abréviation de $(\exists z) \cdot \ y = x + z$.

Ces différentes correspondances sont effectives : on peut passer entre expressions régulières, ensembles semi-linéaires et formules de Presburger en utilisant des algorithmes bien établis (voir [Gin66, GS66a, Reu66, CC01]). Ces constructions sont rappelées dans la première annexe. L'intérêt de la logique de Presburger est sa décidabilité (au contraire de l'arithmétique dans son ensemble, qui est indécidable). Comme pour la décision de la logique du premier ordre, la méthode consiste à construire (par induction sur la structure de la formule) un automate qui reconnait l'ensemble des vecteurs vérifiant la formule (pour un codage particulier des vecteurs en des mots sur un certain alphabet). La validité de la formule se réduit en la non vacuité de l'automate correspondant. Nous utilisons cette construction pour vérifier comme suit la validité d'un arbre. Considérant

un noeud étiqueté par un opérateur ω, on suppose inductivement que chacun de ses sous-arbes immédiats ait été jugé conforme avec une sorte déterminée, on collecte ces différentes sortes pour former un vecteur dont on vérifie qu'il satisfait la formule de Presburger associée à l'image commutative de $\mathcal{L}_\omega = \cup_{s\in\Xi}\mathcal{L}_{\omega,s}$.

Les ensembles semi-linéaires sont par ailleurs clos de manière effective par les opérations booléennes, ce qui fait que, de façon analogue à ce que nous avons exprimé plus haut, nous pouvons décider si le langage d'une grammaire est à commutations près contenu dans celui d'une autre, i.e., la relation $G \leq_c G' \iff L_c(G) \subseteq L_c(G')$ est décidable.

La définition suivante est l'adaptation de la définition 2.3 pour les grammaires à commutations près.

Définition 3.3. *Si $\Xi' \subseteq \Xi$ est un sous-ensemble de l'alphabet des sortes d'une grammaire $G = (\Omega, \Xi, \mathcal{L})$ et $s \in \Xi \setminus \Xi'$, on pose $L(G, \Xi', s) = \{u \in \Xi'^* \mid s \vdash^* u\}$. La projection de G à Ξ' est la grammaire $p_{\Xi'}(G) = (\Omega, \Xi', \mathcal{L}')$ où*

$$\mathcal{L}'_{\omega,s'} = \mathcal{L}_{\omega,s'}[[L(G, \Xi', s)]/s \; ; \; s \in \Xi \setminus \Xi']$$

c'est-à-dire qu'on substitue l'image commutative du langage algébrique $L(G, \Xi', s)$ (qui est donc un langage rationnel par le théorème de Parikh) à la variable non visible $s \in \Xi \setminus \Xi'$ dans la partie droite $\mathcal{L}_{\omega,s'}$ de la règle, dans la grammaire d'origine, associée au symbole visible $s' \in \Xi'$.

Remarque. — Dans la pratique nous n'introduisons pas explicitement le langage régulier $[L(G, \Xi', s)]$ mais on exhibe une expression régulière qui le caractérise à commutations près. Pour cela on utilise le fait que la plus petite solution d'une équation $X = A(X) \cdot X + T$ dans laquelle la variable X n'apparait pas dans T est à commutations près donnée par l'expression régulière $A(T)^* \cdot T$

Exemple 3.4 (voir figure 2)**.**
La projection de la grammaire

$$G = \begin{pmatrix} A \quad \textbf{where} \\ \quad A \to \omega_1\langle BC\rangle \;+\; \omega_2\langle\varepsilon\rangle \\ \quad B \to \omega_3\langle BD\rangle \;+\; \omega_4\langle\varepsilon\rangle \\ \quad C \to \omega_1\langle CA\rangle \;+\; \omega_4\langle CCB\rangle \;+\; \omega_1\langle D\rangle \\ \quad D \to \omega_2\langle AB\rangle \end{pmatrix}$$

sur le sous-alphabet $\Xi' = \{A, B\}$ est donnée par

$$p_{\{A,B\}}(G) = \begin{pmatrix} A \quad \textbf{where} & & \\ \quad A & \to & \omega_1\langle BC\rangle \;+\; \omega_2\langle\varepsilon\rangle \\ \quad B & \to & \omega_3\langle BD\rangle \;+\; \omega_4\langle\varepsilon\rangle \\ \quad C & =_c & CA + CCB + D \\ \quad D & =_c & AB \end{pmatrix}$$

En utilisant $CA+CCB+D =_c (A+CB)C+D$ *on obtient* $C =_c (A+DB)^*D = (A+ABB)^*AB$ *et donc*

$$p_{\{A,B\}}(G) = \left(\begin{array}{lll} A & \textbf{where} & \\ \;A & \to & \omega_1\langle B(A+ABB)^*AB\rangle \;+\; \omega_2\langle\varepsilon\rangle \\ \;B & \to & \omega_3\langle BAB\rangle \;+\; \omega_4\langle\varepsilon\rangle \end{array} \right)$$

L'arbre de la figure 1 (qu'on retrouve à gauche de la figure 2) se présente sous

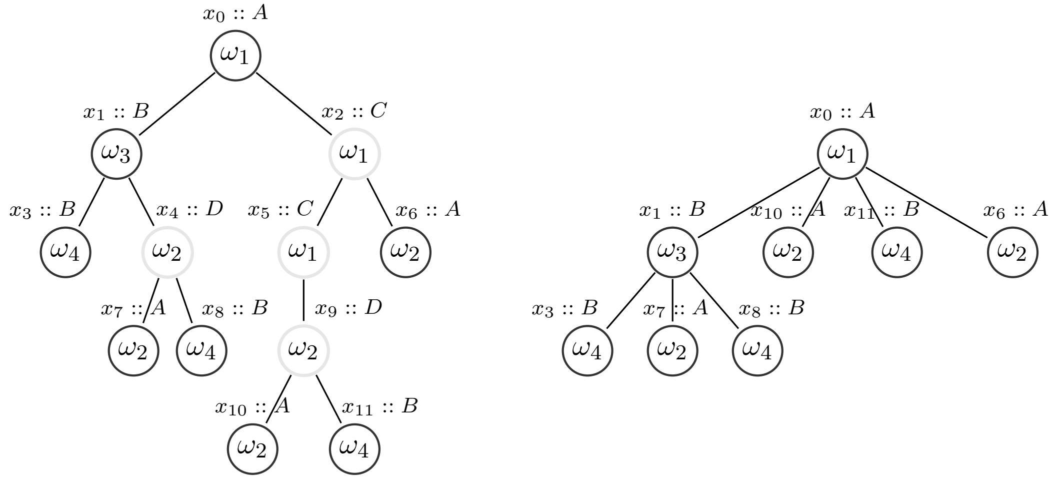


FIG. 3.1 – A gauche, l'arbre t conforme à la grammaire G provenant de la figure 1 : $G \vdash_c t :: A$. A droite, la projection de t sur le sous-alphabet de sortes visibles $\Xi' = \{A, B\}$. On vérifie que $p_{\{A,B\}}(G) \vdash_c p_{\{A,B\}}(t) :: A$.

la forme du système d'équations suivant :

$$\begin{array}{llll} x_0 = \omega_1(x_1, x_2) & x_3 = \omega_4() & x_6 = \omega_2() & x_9 = \omega_2(x_{10}, x_{11}) \\ x_1 = \omega_3(x_3, x_4) & x_4 = \omega_2(x_7, x_8) & x_7 = \omega_2() & x_{10} = \omega_2() \\ x_2 = \omega_1(x_5, x_6) & x_5 = \omega_1(x_9) & x_8 = \omega_4() & x_{11} = \omega_4() \end{array}$$

On peut étiqueter chaque variable par la sorte qui lui est associée lorsqu'on résout ce système dans l'algèbre associée à la grammaire G

$$\begin{array}{llll} x_0^A = \omega_1(x_1^B, x_2^C) & x_3^B = \omega_4() & x_6^A = \omega_2() & x_9^D = \omega_2(x_{10}^A, x_{11}^B) \\ x_1^B = \omega_3(x_3^B, x_4^D) & x_4^D = \omega_2(x_7^A, x_8^B) & x_7^A = \omega_2() & x_{10}^A = \omega_2() \\ x_2^C = \omega_1(x_5^C, x_6^A) & x_5^C = \omega_1(x_9^D) & x_8^B = \omega_4() & x_{11}^B = \omega_4() \end{array}$$

Le système d'équation associé à la projection de l'arbre est obtenu en "effaçant" le symbole d'opérateur en partie droite d'une équation dont la variable en partie

gauche est d'une sorte non visible :

$$\begin{array}{llll} x_0^A = \omega_1(x_1^B, x_2^C) & x_3^B = \omega_4() & x_6^A = \omega_2() & x_9^D = x_{10}^A, x_{11}^B \\ x_1^B = \omega_3(x_3^B, x_4^D) & x_4^D = x_7^A, x_8^B & x_7^A = \omega_2() & x_{10}^A = \omega_2() \\ x_2^C = x_5^C, x_6^A & x_5^C = x_9^D & x_8^B = \omega_4() & x_{11}^B = \omega_4() \end{array}$$

On simplifie ensuite ce système en supprimant toute équation de la forme $x^s = x_1^{s_1} \dots x_n^{s_n}$ *associée à une sorte non visible* $s \in \Xi \setminus \Xi'$ *après avoir substitué la définition de* x_s*, i.e.,* $x_1^{s_1} \dots x_n^{s_n}$*, à chacune de ses occurrences d'utilisation (en fait ici on ne peut avoir qu'une seule telle occurrence car ce système d'équation provient d'un arbre) :*

$$\begin{array}{llll} x_0^A = \omega_1(x_1^B, x_{10}^A, x_{11}^B, x_6^A) & x_3^B = \omega_4() & x_7^A = \omega_2() & x_{10}^A = \omega_2() \\ x_1^B = \omega_3(x_3^B, x_7^A, x_8^B) & x_6^A = \omega_2() & x_8^B = \omega_4() & x_{11}^B = \omega_4() \end{array}$$

Revenons au problème de l'opacité tel que nous l'avons formulé dans l'introduction.

Soit un observateur représenté par sa fonction d'observation $\phi : G \to O$ dans laquelle O est l'ensemble des observations possibles et $\phi(t)$ est l'information que l'observateur a de l'arbre t. Le but de cet observateur est d'inférer de l'information sur t, sachant que l'observateur a une parfaite connaissance du système : il connait la grammaire G décrivant les arbres, il sait par ailleurs comment sa fonction d'observation ϕ est construite et il est en mesure de déterminer l'ensemble $\phi^{-1}(o)$ des arbres qui donnent lieu à une observation donnée $o \in O$. De cette façon, la connaissance qu'il a d'un arbre $t \in G$ est donnée par l'ensemble $\phi^{-1}(\phi(t))$ des arbres donnant lieu à la même observation que celui-ci. Ainsi il sera capable de déterminer qu'un arbre t vérifie la propriété P lorsque l'ensemble $\phi^{-1}(\phi(t))$ est complétement inclus dans P. Ce que traduit la formule suivante :

$$\phi^{-1}(\phi(t)) \subseteq P$$

Lorsque cette relation est vérifiée par un arbre, on dit qu'il y a fuite d'information, ce qui constitue une faille de sécurité dans le cas où la propriété qui a été déduite est une information sensible qu'on souhaitait garder secrète. L'élément t vérifiant la relation ci-dessus est appelé un témoin de la fuite de l'information P.

Une définition informelle de l'opacité serait de dire qu'une propriété P d'un objet est dite *opaque* pour un observateur si celui-ci ne peut déduire que la propriété est satisfaite sur la base de l'observation qu'il a de cet objet. De façon plus formelle

Définition 3.5. *Une propriété secrète est opaque sur G pour un observateur si pour tout observation d'un arbre de G satisfaisant la propriéte P,* $t \in P$*, il existe un autre arbre t', ayant la même observation :*

$$\forall t \in P \; \exists t' \notin P \; \phi(t) = \phi(t')$$

De façon équivalente, la propriété P est opaque vis-à-vis de ϕ si et seulement si $\phi(P) \subseteq \phi(U \backslash P)$

Etant donné l'ensemble C des arbres reconnnus par la grammaire $\mathcal{G} = (\Omega, \Xi, \mathcal{L})$. Soit $\Xi' \subseteq \Xi$ un sous-ensemble de sortes visibles. Les observations partielles sont données par la fonction d'observation $\phi_{\Xi'}$, et pour chaque observateur, une propriété des arbres représentée par un automate d'arbres A_S qui est appelée secret et est notée S. On s'interroge sur la possibilité d'un observateur d'inférer qu'un arbre du système se trouve dans le secret, à partir de la vue partielle qu'il a de cet arbre. La figure suivante illustre les propos ci-dessus :

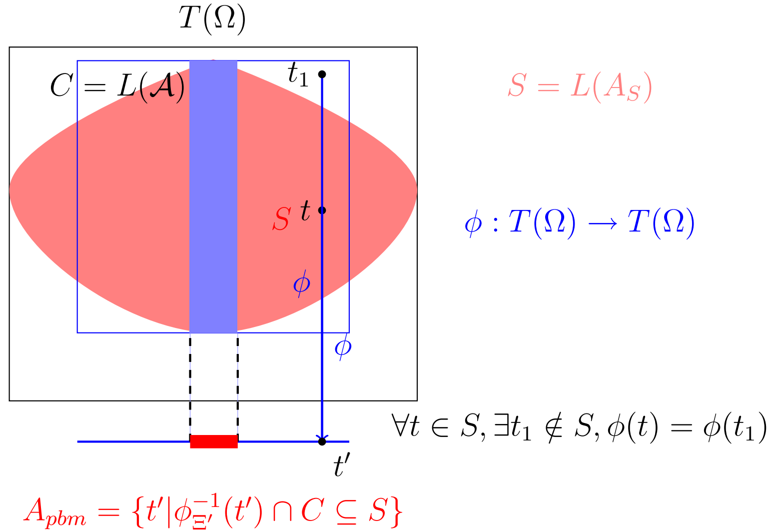


FIG. 3.2 – Opacité dans les systèmes workflows

L'ellipse rouge décrit l'ensemble des secrets $S = L(A_S)$, le carré bleu décrit notre système, $C = L(\mathcal{A})$ et le grand carré noir représente l'ensemble des arbres construit sur l'alphabet Ω. La bande bleue représente les arbres pour lesquels le système n'est pas opaque. Le trait bleu horizontal représente les arbres projetés, le petit rectangle rouge sur cette bande représente A_{pbm}.

A_{pbm} décrit la projection des arbres contre-exemples, i.e des arbres pour lesquels le système n'est pas opaque. Le secret S est opaque vis-à -vis de ϕ si et seulement si A_{pbm} est vide, c'est à dire que $\phi_{\Xi'}(S \cap G) \leq_c \phi_{\Xi'}(G \setminus S)$. L'algorithme suivant est proposé pour vérifier l'opacité d'un système donné en entrée.

Algorithme de vérification de l'opacité

Entrées : C, S : automate d'arbres finis déterministes, ϕ

1 Calculer $C \cap S$
2 Calculer $C \backslash S$
3 Résoudre le système d'équation pour les symboles de $\Xi \backslash \Xi'$ pour la grammaire associée à $C \cap S$. Soit $\phi_{\Xi'}(G)$.
4 Résoudre le système d'équation pour les symboles de $\Xi \backslash \Xi'$ pour la grammaire associée à $C \backslash S$. Soit $\phi_{\Xi'}(G^{'})$.
5 Calculer $F = \phi_{\Xi'}(G) \backslash \phi_{\Xi'}(G^{'})$ et vérifier la vacuité.

Théorème 3.6. *L'opacité des artefacts est un problème NP-complet.*

Démonstration. Les arbres à arité variable non ordonnées sont clos par opération booléennes car les opérations booléennes sont compatibles avec la représentation binaire des arbres à arité variables ordonnés [CDG+08]. Par hypothèse de commutativité cette famille est également close par les projections ϕ associées à un sous ensemble de sortes visibles. Les automates d'arbres C et S étant déterministes, la compléxité des calculs de la différence $C \backslash S$ et de $C \cup S$ sont polynomial en temps. Le calcul de $\phi_{\Xi'}(X)$ est linéaire en la taille des symboles non-visibles $\Xi \backslash \Xi'$ donc polynomial également. Tester la vacuité de l'automate $\phi_{\Xi'}(S \cap C) \backslash \phi_{\Xi'}(C \backslash S)$ est équivalent au problème de l'inclusion des automates d'arbres non déterministes $\phi_{\Xi'}(S \cap C) \subseteq \phi_{\Xi'}(C \backslash S)$ qui est exponentielle. Pour la complétude, on utilise le fait que l'intersection non vide d'automates d'arbres et d'automates d'arbres déterministes est EXPTIME-complete [Vea97] et on réduit ce problème au problème de l'opacité de la manière suivante. Soit $T(\Omega)^\sharp$ l'ensemble des arbres construits au dessus de $\Omega \cup \{\sharp\}$, où $\sharp$ est un nouveau symbole. Si $L \subseteq T(\Omega)$, on lui associe le langage $L^\sharp = \{\sharp(t) \mid t \in L\}$. Soit $\mathcal{A}_1$ un automate d'arbres déterministe et $\mathcal{A}_2$ un automate d'arbres non-déterministes. $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$ ssi $L(\mathcal{A}_1)^\sharp \subseteq L(\mathcal{A}_2)^{\sharp c}$. On considère le problème d'opacité associé à $C = L(\mathcal{A}_1)^\sharp$, $S = L(\mathcal{A}_2)^{\sharp c}$ et pour lequel l'ensemble des symboles visibles $\Xi' = \{\sharp\}$ est réduit au seul symbole $\sharp$. Pour tout arbre $t \in T(\Omega)^\sharp$, on a $\phi^{-1}(t) = T(\Omega)^\sharp$ et par conséquent le système est opaque ssi $C \subseteq S$ i.e. $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) = \emptyset$. □

Lorsque le système n'est pas opaque, i.e lorsqu'on a testé la vacuité de l'automate $\phi_{\Xi'}(S \cap C) \backslash \phi_{\Xi'}(C \backslash S)$ et qu'on a vu que le résultat était différent du vide, il faut rechercher des solutions pour rendre le système opaque. La première solution est de faire du contrôle, i.e enlever tous les arbres qui rendent le système non opaque. Cette technique classique utilisée dans les systèmes à évènements discrets [JPH10] consiste à restreindre le comportement du système de façon minimale afin d'en restaurer l'opacité. Cette solution produit une grammaire qui génère exactement les documents de C pour lesquels aucun secret n'est dévoilé. Le résultat peut néanmoins être peu satisfaisant d'un point de vue pratique si cela peut forcer à exclure trop de documents, et en particulier des documents que le concepteur du système souhaiterait conserver dans son modèle.

Une autre méthode moins invasive consiste à restructurer la grammaire de sorte à rajouter un minimum de bruit pour assurer l'opacité. Cette méthode repose sur les techniques d'évolution de schéma XML préservant la validité des documents. Nous ne disposons pas actuellement d'une solution totalement satisfaisante dans ce dernier cas. Nous présentons néanmoins ci-dessous les résultats partiels obtenus.

Assurer l'opacité par contrôle

Comme illustré à la figure 3, les arbres qui n'assurent pas l'opacité du système (représentés par la bande bleue) correspondent à la projection inverse de $\phi(S \cap C)\backslash\phi(C\backslash S)$.

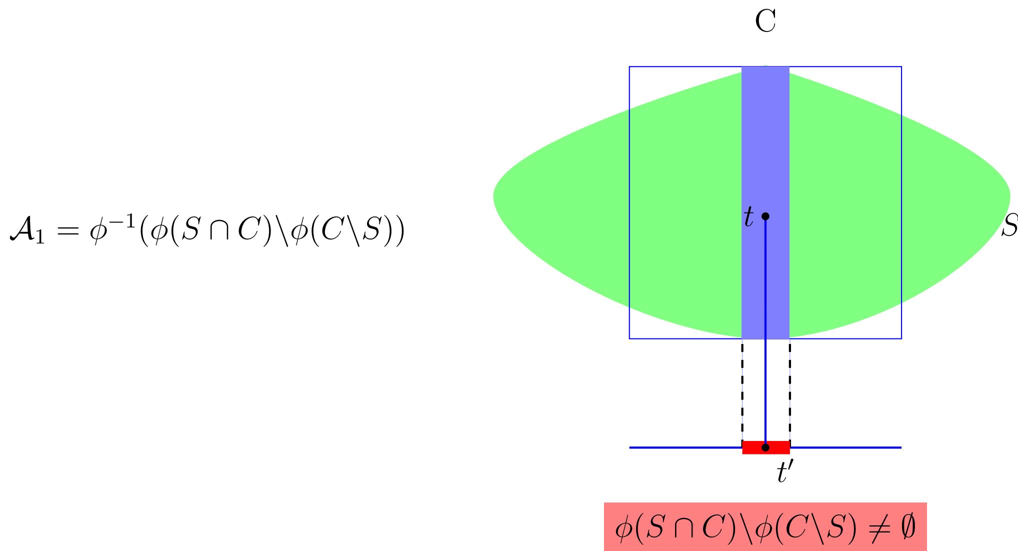


Fig. 3.3 – Assurer l'opacité par contrôle

$\mathcal{A}_1$ un automate reconnaissant ce langage. Il suffit alors de remplacer le système $\mathcal{C}$ (l'automate reconnaissant C) par $\mathcal{C} \cap \neg\mathcal{A}_1$.

Entrée : $\mathcal{A}_1 = \phi^{-1}(\phi(S \cap C)\backslash\phi(C\backslash S))$, ϕ, $\mathcal{C}$
Problème : Calculer $\mathcal{C}' = \mathcal{C} \cap \neg\mathcal{A}_1$.

Complémenter un automate nécessite au préalable de le déterminiser ce qui peut se faire en temps exponentiel en la taille de l'automate. Si on décide de représenter les systèmes par des automates déterministes alors leurs complémentations se

calcule en temps linéaire néanmoins le résultat ($\neg\mathcal{A}_1$ et donc aussi $\mathcal{C} \cap \neg\mathcal{A}_1$) seront non-déterministes et nous devrons tout de même procéder à une étape de déterminisation. Ainsi

Théorème 3.7. *Le problème de contrôle de l'opacité se résout en temps exponentiel.*

Assurer l'opacité par rajout de bruit

Dans cette section on présente une méthode alternative pour assurer l'opacité. Le but est d'étendre la grammaire associée à notre système G_C en y rajoutant un minimum d'arbres afin d'en assurer l'opacité.

Le principe est d'appliquer les techniques d'évolution de schéma XML préservant la validité des documents[BDA$^+$04b, BDA$^+$04a] sur la grammaire G_C pour obtenir une nouvelle grammaire $G_{C'}$. Les images inverses des arbres t_i appartenant à $L(A_{pbm})$ sont considérés comme des mis-à-jour pour notre système G_C. Ces images inverses sont données par les langages $L(A_{t_i}) = \phi^{-1}(t_i)\backslash S$. Les arbres rajoutés de cette manière peuvent ne pas appartenir au secret et ont même projection que les arbres qui violent le secret. La méthode est illustrée à la figure 3

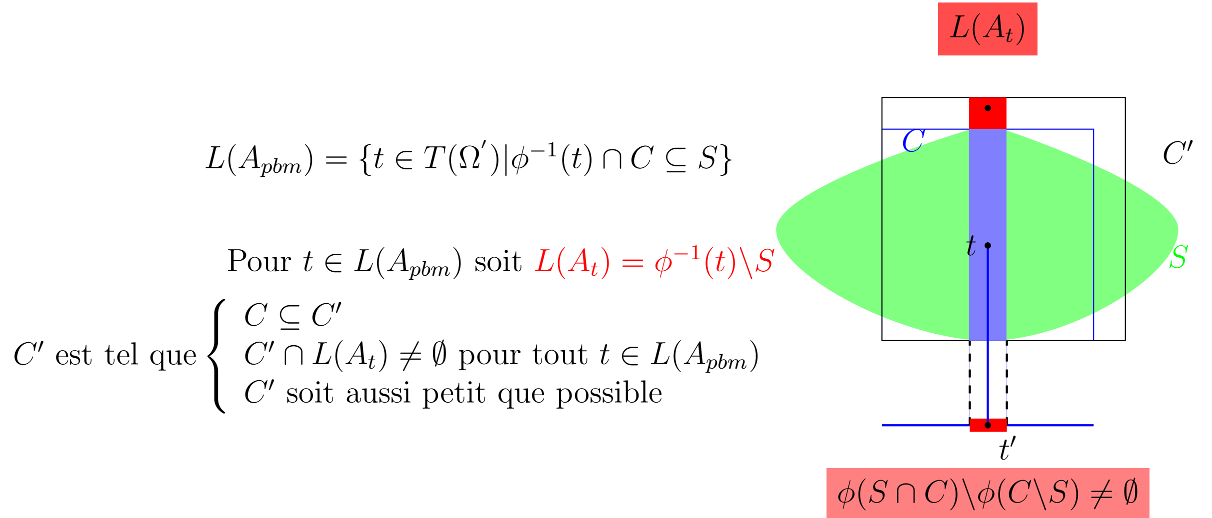


Fig. 3.4 – Assurer l'opacité par rajout de bruit

On sait résoudre ce problème, si $L(A_{pbm})$ est fini et les ensembles $L(A_t)$ correspondants son non vides, en utilisant des techniques d'extensions conservatives de schémas XML [CZ00, BDA$^+$04b, BDA$^+$04b, Dua05]. Notre idée lorsque $L(A_{pbm})$ est infini (ce qui sera généralement le cas) est de se ramener à un nombre fini de

cas en utilisant l'ordre de plongement sur les arbres [Kru60] : $t \leq t'$ ssi $t = a(T)$, $t' = b(T')$ avec $a = b$ et il existe une injection $j : T \to T'$ telle que $\forall t \in T \;\; t \leq j(t)$. Cet ordre est un bel ordre, c'est-à-dire que tout ensemble d'arbres contient un nombre fini d'arbres minimum (dans cet ensemble) pour l'ordre de plongement. Il faudrait donc se restreindre aux extensions de grammaires telles que si $t \in C' \setminus$ est tel que $\phi^{-1}(\phi(t) \cap C \subseteq S$ alors on pourra trouver un relévement $t'' \in C' \setminus S$ de t' (i.e., $\phi(t'') = t'$) pour tous les $t' \in A_{pbm}$ dans lesquels t se plonge : $t \leq t'$. Il faudrait par ailleurs vérifier que de telles extensions existent dès que $L(A_t) \neq \emptyset$ pour tous les $t \in L(A_{pbm})$.

Chapitre 4

Implémentation

Dans ce chapitre, on s'intéresse à l'implémentation et aux tests des algorithmes proposés dans la thèse. Il est structuré en trois parties. Dans la première partie on présente l'outil qui implémente l'arithmétique de Presburger. Il sagit de *TAPAS* pour *TAlence Presburger Arithmetic Suite*, développé en langage C au sein du laboratoire Bordelais (*LABRI*).

Dans le seconde partie on présente l'interface *FFI* (Foreign Function Interface) de Haskell qui nous permet d'appeler des fonctionalités de TAPAS à partir de notre code développé en Haskell.

Dans la troisième partie nous présentons notre code Haskell pour la reconnaissance des arbres modulo commutation, l'implémentation des grammaires étendues, la conformité d'un arbre selon une grammaire, le calcul de la grammaire projetée et enfin la vérification de l'opacité.

4.1 – Implémentation de l'arithmétique de Presburger : TAPAS

Ainsi que nous l'avons décrit dans l'annexe précédente (Chapitre 4) la reconnaissance des ensemble semi-linéaires utilise un encodage binaire et des automates manipulant de telles suites de vecteurs de bits. Parmi ces outils, on trouve *LASH*[1], pour *Liège Automata-based Symbolic Handler*, développé à l'université de *LIEGE* par *Bernard Boigelot*. Dans *LASH* on trouve différentes librairies comme la *NDD* (Number Decision Diagram). Elle contient un système de représentation d'un ensemble de vecteur d'entiers pouvant être utilisée pour représenter l'ensemble des solutions d'une formule de Presburger. Malheureusement cet outil n'est pas maintenu et ne fonctionnne pas sur les architectures 64 bits, car les types qui y sont définis dépendent de l'architecture. Un autre outil étudié pendant cette

[1] www.montefiore.ulg.ac.be/ boigelot/research/lash/

thèse est le package Omega [Pug92] qui est problablement l'outil Presburger le plus utilisé. Il est écrit en C mais n'est malheureusement pas aussi efficace que *LASH*.

L'outil qu'on a finalement choisi est *TAPAS* (pour Talence Presburger Arithmetic Suite)[2]. C'est un ensemble de librairies écrite en C pour l'arithmétique de Presburger, accompagné d'un petit solveur. En plus de celles qui sont listées ci-dessous comme *MONA*, *LIRA*, *PRESTAF*, *PPL*, *TAPAS* fait appel à la plupart des librairies disponibles sur l'arithmétique de Presburger, C'est ce qui le rend ainsi complet.

Le paragraphe qui suit détaille un peu plus l'utilisation de la librairie *TAPAS*. La plupart des données de cette partie proviennent du cite de *TAPAS*.

4.1.1 – TAPAS : Talence Presburger Arithmetic Suite

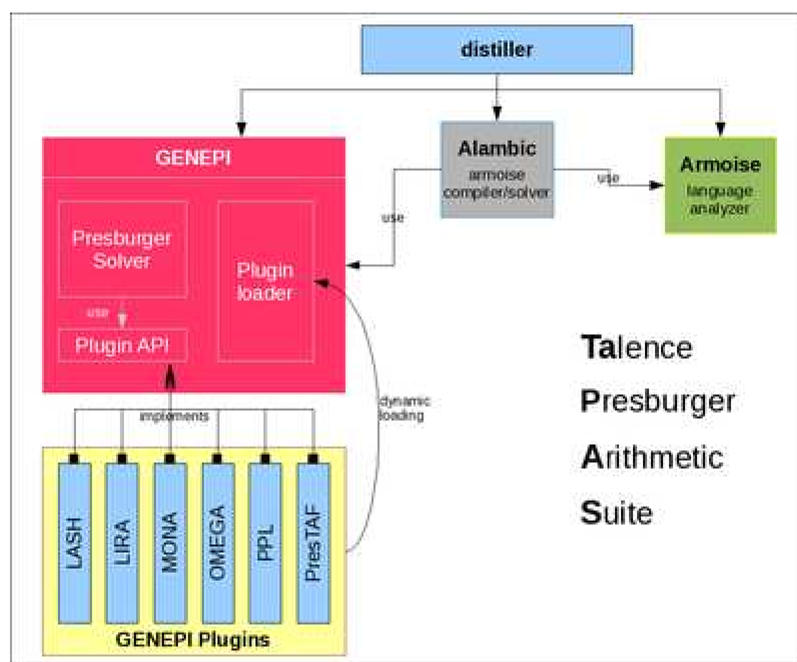


FIG. 4.1 – Architecture TAPAS tirée du site web de TAPAS

Comme indiqué à la figure 4 *TAPAS* est composée de plusieurs outils. L'un des principaux, *GENEPI*, est une interface de programmation générique qui peut être utilisée comme solveur de formule de Presburger. *GENEPI* fait appel à des procédures qui sont importées en chargeant dynamiquement des plugins. Ces plugins sont répartis en fonction de la nature des nombres manipulés :

[2]tapas.labri.fr/trac

1. **Plugins sur les nombres naturels :**
   a. **La librairie *DFA***(*Data-structure automaton*) de *MONA* est utilisée pour encoder l'automate binaire.
   b. **La structure de données *Shared Automata*** est utilisée pour implémenter le plugin *PresTAF*. *PresTAF* est le solveur/synthétiseur basé sur cette structure de données. Elle permet de synthétiser une formule de Presburger (utilisant le langage *Armoise* présenté plus loin) à partir d'un automate binaire.
2. **Plugins sur les entiers**
   a. **LASH** (*Liège Automata-based symbolic Handler*) est utilisé pour implémenter un solveur sur les entiers. La librairie *NDD* (Number Decision Diagrams) est utilisée pour l'implémenter.
   b. **Omega** est l'un des plugins utilisés qui n'est pas basé sur l'utilisation des automates binaires.
3. **Plugins sur les nombres réels**
   a. **PPL**(*Parma Polyhedra Library*) est un autre plugin qui n'est pas basé sur l'utilisation des automates binaires.
4. **Plugins sur les nombres réels et entiers**
   a. **LIRA** (*Linear Integer/Real Arithmetic Solver*) est un plugin basé sur une déterminisation faible des automates de Büchi.
   b. **RVA** (*Real Vector Automata*) est utilisée pour implémenter un solveur sur les nombres réels et entiers.

GENEPI présente plusieurs avantages. D'un côté, les applications qui nécessitent la résolution de formules de Presburger peuvent profiter de l'abstraction faite sur les solveurs, qui sont chargés dynamiquement et donc, de reporter le choix du solveur effectivement utilisé au moment de l'exécution. De l'autre côté, les développeurs de solveurs peuvent utiliser n'importe quelle application basée sur *GENEPI* afin de comparer leur solveur à d'autres solveurs.

4.1.2 – Le Langage Armoise

Le langage *Armoise* est utilisé pour décrire l'ensemble des vecteurs d'entiers et de réels. Un programme *Armoise* est une suite d'ensembles de vecteurs de la forme :

Set 1 ;
Set 2 ;
...
Set n ;

Chacun de ces ensembles peut être décrit de différentes manières :

. comme un ensemble prédéfini ou un ensemble de constantes,

. par une formule donnant une relation entre les variables,
. en utilisant les opérations sur les ensembles existants.

Dans *Armoise*, il est possible de déclarer localement des ensembles comme suit :
let
S1 := Définition de S1 ;
S2 := Définition de S2 ;
. . .
Sn := Définition of Sn ;
in
Définition de l'ensemble principal S ;

Chaque S_i est un ensemble et peut être utilisé dans la définition de l'ensemble principal S.

Exemple 4.1.
let
$S1 := \{(x, y) \in \mathcal{N}^2 \mid \exists\, z \in \mathcal{N}\; x = y + z\};$
in
$\{(x, y) \in \mathcal{N}^2 \mid (x, y) \in S1\};$

Il existe cependant des ensembles prédéfinis :
. **nat** est l'ensemble des entiers naturels (c'est-à-dire des entiers positifs).
. **int** est l'ensemble des entiers (positifs et négatifs).
. **posi** est l'ensemble des nombres réels positifs ou nuls.
. **real** est l'ensemble des nombres réels (positifs et négatifs).

Exemple 4.2. *l'ensemble S1 ci-dessous se récrit de la manière suivante :*

$$S1 := \{(x, y)\; in\; (nat, nat) \mid \exists\, z\; in\; nat\; x = y + z\};$$

Les intervalles peuvent être aussi définis dans Armoise de la façon suivante :
. $[\min \cdots \max]$ définit l'intervalle des nombres réels compris entre min et max (bornes incluses) ;
. $[\min, \max]$ définit l'ensemble des entiers compris entre min et max (inclus).

Un ensemble est alors caractérisé par une formule **v in Dom | F ;** dans laquelle :
. **v** est un vecteur de variables : e.g. a, (x, y, z) ou (a, b, (c, d))
. **Dom** définit le domaine de v, e.g. (nat, real, (real, nat)).
. **F** est une formule qui décrit les relations entre les variables de v.

On peut avoir accès à chaque composante de v par *v[indice]* où indice varie entre 0 et la taille de v moins 1.

Une formule Armoise est construite en utilisant les constantes booléennes, les opérations booléennes ou la comparaison de termes :
1. Constantes booléennes : *true* et *false*

2. les opérations booléennes sont :
   a. la disjonction F1 | F2, F1 or F2
   b. la conjonction F1 & F2, F1 and F2
   c. l'equivalence F1 <=> F2
   d. l'implication F1 => F2
   e. la négation ! F, not F
3. Comparaison de termes :
   T1 < T2, T1 <= T2, T1 > T2, T1>= T2, T1 = T2 and T1 ! = T2
4. Les quantificateurs :
   a. exists x1 , ..., xn ((x1 , ..., xn) in Dom and G)
   b. forall x1 , ..., xn ((x1 , ..., xn) in Dom and G)

Ces termes sont à leur tour construits à partir de constantes entières, de vecteurs et d' opérations arithmétiques :

a. les constantes entières positives : 3, 14, 15, ...
b. les vecteurs : $(t_1, t_2, \cdots, t_n)$
c. les éléments d'un vecteur : $v[i_1]\cdots[i_k]$ où i_j sont des constantes entières.
d. les opérateurs arithmétiques :
   i. l'addition t1 + t2.
   ii. la soustraction t1 - t2.
   iii. la multiplication t1 * t2.
   iv. la division t1 / t2.
   v. le modulo t1 % t2.
   vi. l'opposé -t

Ces opérations peuvent être étendues aux vecteurs. Par exemple $2 * ((1, 2 * t) + (x, y))$ désigne le vecteur $(2*x+2, 2*y+4*t)$. On peut aussi définir des opérations sur les ensembles :

. Union S1 || S2 ;
. Intersection S1 && S2 ;
. Différence S1 \ S2 ;
. Différence symétrique $S1\hat{}S2$ (pour $(S1\backslash S2)$ || $(S2\backslash S1)$) ;
. Compléments dans les nombres réels !S (pour real \ S).

Les opérations numériques sur ces ensembles sont données par *S1 'op' S2* pour { x | exist x1 , x2 ((x1 , x2) in (S1 , S2) and x = x1 op x2) } où op appartient à { +, -, *, /, % }. (S1 ,$\cdots$, Sn) représente le produit cartésien des ensembles S_i.

Nous allons traiter l'exemple donné au chapitre 1. Pour rappel on avait la formule suivante : $\exists\ \pi = (x_1, x_2, x_3, x_4) \in N^4$

$$\Psi(y_1, y_2) = \Delta' \Leftrightarrow \begin{cases} x_2 \geq 1 \\ x_3 - x_2 = -1 \\ x_2 - x_3 = 1 \\ x_4 = x_1 + 2 * x_2 - x_3 \end{cases}$$

pour laquelle on aimerait trouver les solutions. Ce système d'équation écrit en Armoise donne :

$$\{x \text{ in } (nat, nat, nat, nat, nat) \quad | \quad \begin{array}{l} x[1] \geq 1 \text{ and} \\ x[2] - x[1] = -1 \text{ and} \\ x[1] - x[2] = 1 \text{ and} \\ x[3] = x[0] + 2 * x[1] - x[2]\} \end{array}$$

Pour résoudre ce système on doit faire appel au compilateur de *Armoise* qui s'appelle *Alambic*. Le but d' *Alambic* est d'appliquer plusieurs règles de récriture sur des formules *Armoise* de sorte à rendre calculable la formule proposée en utilisant *GENEPI*. Enfin on a le solveur en ligne de commande *distiller* qui rassemble *GENEPI*, *Armoise* et *Alambic* qui calcule la solution du système. *Distiller* est distribué dans le paquet *Alambic*.

PresTAF, qui est un plugin de *GENEPI*, calcule l'automate correspondant à la formule donnée en entrée et précise à la fin de l'exécution le nombre d'états nécessaires à la construction de l'automate.

Remarque. — Les équations redondantes sont éliminées dans la formule générée par *Distiller*

En exécutant la commande *distiller -s* (pour solutions) sur notre système d'équations on a le résultat suivant :

```
mdiouf@localhost:~
File Edit View Search Terminal Help
[mdiouf@localhost ~]$ echo "{x in (nat,nat,nat,nat)| 1 * x[1 ]>=1 and 1 * x[2] -
 1 * x[1] = -1 and 1 * x[1] - 1 * x[2] = 1 and x[3]= x[0] + 2 * x[1] + 1*x[2] };
" | distiller -s
let
  G0 := int * (1, 0, 0, 1) + int * (0, 1, 1, 3);
  nat4 := (nat, nat, nat, nat);
  R := (0, 1, 0, 2) + G0;
in
  { (x_0, x_1, x_2, x_3) in nat4 | (x_0, x_1, x_2, x_3) in R };
// Automata size             : #SA=4 #EA=4 #S=39 #D=3 #Refs=3 #to0=3/78
// Formula size              : 56 nodes
// Time to synthesis formula : 0 ms
[mdiouf@localhost ~]$
```

Fig. 4.2 – Résultat de l'exécution de la commande Distiller

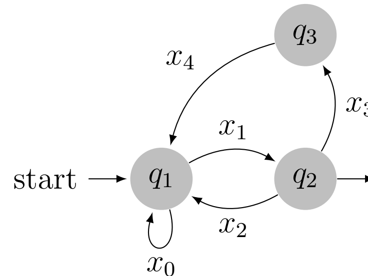
Le résultat indique que l'ensemble des solutions est de la forme :

$$R := (0,1,0,2) + int * (1,0,0,1) + int * (0,1,1,3);$$

où *int* est n'importe quel entier, en l'occurence si int prend la valeur 1, le vecteur v suivant appartient à R :

$$v := (0,1,0,2) + (1,0,0,1) + (0,1,1,3) := (1,2,1,6) = (x_0, 2x_1, x_2, x_3);$$

Ce qui veut dire que v est bien l'image commutatif du chemin acceptant $x_0x_1x_2x_1$ comme on peut le voir dans la figure ci-dessous :



Si $\mathcal{L}$ représente le langage $A^*(AB)^*$, la clotûre de $[\mathcal{L}]$ modulo commutation est générée par l'ensemble R.

4.2 – Foreign Function Interface : FFI

Pour pouvoir utiliser *TAPAS* à l'intérieur de code Haskell, on utilise à l'interface *Foreign Function Interface (FFI)* d'Haskell. Cette interface permet de communiquer avec des programmes écrits dans d'autres langages de programmation, avec une gestion des exceptions et des problèmes d'incompatibilité qui peuvent surgir lors de la programmation avec cette interface, liés essentiellement aux spécificités du langage cible comme le typage.

Pour ce faire, on a besoin de comprendre la convention utilisée pour les appels de fonctions, les systèmes de types, les structures de données, le mécanisme d'allocation de mémoire et la stratégie de liaison. Cette correspondance n'est pas chose facile puisqu'il faut avoir des connaissances dans le langage cible. On décrit dans cette section le fonctionnement de cette interface de programmation.

4.2.1 – Spécificité de FFI

FFI est une extension de Haskell 98 [3] permettant d'utiliser un programme écrit dans un autre langage qu'Haskell dans Haskell et inversement. Dans sa

[3] www.haskell.org/definition/

version 1.0 [Cha], elle donne une spécification complète de l'interaction entre un code Haskell et un code écrit en langage C, qui s'effectue par l'appel de code C dans Haskell. Elle permet aussi de prendre en compte l'appel de code écrit en C++ ou en java. Sa spécification est indépendante du langage cible et présente deux domaines pour lesquels *FFI* est considérée comme un langage spécifique :

– la spécification de noms externes qui dépend de la convention utilisée pour l'appel de fonctions externes comme nous allons le voir dans la section suivante,
– la correspondance des types be base du langage cible : par exemple le type entier du langage C :`Int` peut être codé sur 16, 32 ou 64 bits, *Haskell FFI* introduit un nouveau type `CInt` avec le même codage.

On se restreint aux aspects de *FFI* utiles au développpement de notre interface. Pour plus de détails sur l'utilisation de *FFI*, voir [OSG08].

FFI ne peut appeler que des fonctions C et des variables globales. Il n'existe pas d'autres mécanismes pour accéder à d'autres entités d'un programme C comme les constantes définies avec le mot clé `#define`.

Pour appeler une fonction C, on a besoin de trois choses : connaître le nom de la fonction, son type et le fichier d'entête qui lui est associé. Ensuite pour effectuer l'appel on utilise le mot clé `foreign import` comme ceci :

```
foreign import ccall unsafe "ccl-init.h ccl_init"
                  c_ccl_init :: IO()
```

Cela définit une nouvelle fonction `c_ccl_init` en C à travers la fonction `ccl_init` qui se trouve dans le fichier `ccl-init.h`.

Le mot clé `ccall` permet d'invoquer `ccl_init` lorsque `c_ccl_init` est appelée. Haskell passe le contrôle au programme C, et le résultat est retourné à Haskell comme une valeur de type `IO()`.

Lorsqu'on écrit la correspondance, on a besoin de convertir les types C en types *FFI*. Par exemple le type `void` en C corresponds à `()`. La plupart des types primitifs de C sont représentés dans *FFI* avec des noms intuitifs comme `CInt`, `CDouble`, `CString`.

Remarque. — Les fonctions C retournent le plus souvent une valeur indiquant le statut d'exécution de la fonction, ou le type void mais très rarement le résultat de la fonction. Ceci est dû aux effets de bords des fonctions C qui peuvent retourner, pour une valeur d'entrée donnée, différentes valeurs de sortie selon que la valeur d'entrée est globale, locale ou autre. Pour pallier ce problème on utilise la monade d'Entrées-Sorties afin de capturer les effets de bords.

4.2.2 – TAPAS dans Haskell

On retrouve dans le fichier `distiller.c` toutes les fonctions utiles au calcul de l'ensemble des solutions d'une formule de Presburger. Mais il n'est pas possible d'importer le fichier .C directement dans *FFI* et l'exécuter. Il faut transcrire la fonction qui permet de faire le calcul d'une formule de Presburger en *FFI*, ainsi que toutes les fonctions qui dépendent du calcul du résultat. La première étape consiste ainsi à transcrire la fonction `alambic_compute_predicate`

```
extern genepi_set *alambic_compute_predicate
        (genepi_solver *solver,
         const armoise_normalized_predicate *P,
         alambic_error *p_error);
```

Cette fonction se trouve dans `alambic.h`. Elle prend comme entrée un solveur, une formule de Presburger normalisée et une variable qui retourne le statut de l'exécution et retourne un ensemble genepi.

On a besoin pour transcrire la fonction de faire correspondre le type de cette fonction avec ceux d'Haskell. Or, on remarque qu'aucun de ces types n'est un type standard de C, hormis le fait qu'ils soient des pointeurs.

Le type pointeur est donné en Haskell par `Ptr`. On retrouve dans ce type toutes les opérations qui existent déjà dans son homologue en C (allocation, déréférencement, modification, pointeur null :`nullptr`).

Il reste à représenter les types abstraits `genepi_solver`, `alambic_error`, `const armoise_normalized_predicate` ainsi que la valeur retournée `genepi_set`. Puisque les types en C sont traités de manière abstraite, on peut choisir n'importe quel type Haskell pour tout type arbitraire. En l'occurence on peut choisir un type portant le même nom qui pointe vers `()`. Soit

```
type GENEPI_SOLVER = ()
type GENEPI_SET = ()
type ARMOISE_NORMALIZED_PREDICATE = ()
```

Le type `ALAMBIC_ERROR` est une énumeration d'entiers `Int`, on lui associe par conséquent le type de base `CInt` en Haskell.

On peut maintenant faire l'appel de la fonction C `alambic_compute_predicate` à partir d'Haskell en utilisant le mot clé `foreign import`. Cette fonction retourne une valeur de type `Input/Output (IO)`, puisque le pointeur retourné varie à chaque appel.

```
foreign import ccall unsafe "alambic.h alambic_compute_predicate"
        c_alambic_compute_predicate :: Ptr GENEPI_SOLVER
```

```
                      -> Ptr ARMOISE_NORMALIZED_PREDICATE
                      -> Ptr CInt
                      -> IO (Ptr GENEPI_SET)
```

Ainsi, les types de données sont des objets opaques,inconnus, que l'on traite comme des pointeurs sur `()`. Lorsque l'on fait une déclaration `foreign import`, on peut optionnellement utiliser le mot `safe` pour plus de sécurité dans les appels C mais il reste moins efficace et donc on utilisera plutôt le mot clé `unsafe`.

Il reste la question de la gestion de la mémoire allouée à ces pointeurs abstraits retournés par les librairies de C. Il se trouve qu'on a besoin de leur allouer de la mémoire, ceci est géré du coté du programme C. Mais il peut arriver parfois qu'on est besoin de désallouer cette mémoire

On peut utiliser le gestionnaire de mémoire d'Haskell pour désallouer automatiquement la mémoire qui n'est plus utilisée. Pour ce faire, on utilise le type `ForeignPtr` et la fonction *newForeignPtr* qui utilise le gestionnaire de mémoire d'Haskell. Voici le type de cette fonction

```
newForeignPtr :: FinalizerPtr a -> Ptr a -> IO (ForeignPtr a)
```

Cette fonction prend deux arguments : un `finalizer` pour l'exécution lorsque les données ne sont plus utiles et un pointeur associé à la structure de données C. Il retourne un nouveau gestionnaire de mémoire qui libérera la mémoire lorsque le gestionnaire de mémoire aura décidé qu'elle n'est plus utile.

Ceci nous permet de retourner un type de données exploitable par d'autres fonctions ou visible par l'utilisateur.

```
data Solution = Solution !(ForeignPtr GENEPI_SET)
               ! CInt
             deriving (Eq, Ord, Show)
```

Ce nouveau type de données `Solution` est divisé en deux parties. La première permet de gérer la mémoire de la structure de données `GENEPI_SET` allouée dans C. La seconde partie permet d'avoir le statut d'exécution de la fonction.

On a choisi `Either` puisque la fonction peut optionnellement retourner une erreur si le `Solver` et/ou le prédicat retourné n'est pas valide. La fonction complète qui résout une formule de Presburger est la suivante :

```
compute_predicate :: Solver -> Ctree -> CInt -> IO (Either CInt Sol)
compute_predicate (Solver solver _) (Ctree nptree _) n =
    withForeignPtr solver $  \ solv -> do
      withForeignPtr nptree $ \ npt -> do
          alam <- new n
```

```
          z <- peek npt
          let obj = treeObject z
          let pred = armp obj
          let npred = anp pred
          solut <- c_alambic_compute_predicate solv npred alam
          t <- peek alam
          if t /= 0
             then do
               return (Left 0)
             else do
                if solut==nullPtr
                     then do
                       err <- peek alam
                       return (Left alam)
                     else do
                       solut <- newForeignPtr finalizerFree solut
                       return (Right (Solution solut n))
```

L'entrée de cette fonction est un `Solver` retourné par la fonction `c_genepi_solver_create` qui crée un `genepi_solver` à partir d'un plugin.

```
foreign import ccall unsafe "genepi.h genepi_solver_create"
c_genepi_solver_create :: Ptr GENEPI_PLUGIN
                          -> GenepiFlag
                          -> CInt
                          -> CInt
                          -> IO (Ptr GENEPI_SOLVER)
```

un `Ctree` qui représente la formule de Presburger sous forme d'arbre de prédicats.
Elle est donnée par la fonction

```
foreign import ccall unsafe "alambic.h alambic_normalize_predicate_tree"
        c_alambic_normalize_predicate_tree :: Ptr ARMOISE_CONTEXT
                                              -> Ptr CCL_TREE_ST
                                              -> Ptr CInt
                                              -> IO (Ptr CCL_TREE_ST)
```

Elle normalise l'arbre au préalable avant de le passer à la fonction `compute_predicate`.
L'entier `n` qui permet de retourner le statut d'exécution de la fonction. Ensuite on récupère les valeurs des pointeurs retournés par les deux fonctions ci-dessus. Nous

utilisons pour cela la méthode `withForeignPtr`. La fonction `compute_predicate` prend en entrée un `CInt` alors que `c_compute_predicate` a besoin d'un `Ptr CInt`, la fonction *new* permet cette transformation. Son type est :

```
new :: Storable a -> a -> IO (Ptr a)
```

Cette fonction alloue d'abord de la mémoire à son argument et le stocke dessus. Dans la structure de données de l'arbre de predicats, la fonction `c_alambic_predicate` a besoin de récupérer le champ `Object` qui est de type `Ptr ()` qu'on doit convertir en type `Ptr ARMOISE_NORMALIZED_PREDICATE`. Pour cela on commence par instancier la classe `Storable` pour stocker toute la structure de données du type `CCL_TREE_ST` dans Haskell comme suit :

```
instance Storable CCL_TREE_ST where
  sizeOf _ = (#size ccl_tree)
  alignment _ = alignment (undefined :: CInt)
  peek ptr = do
      refc <- (#peek ccl_tree, refcount) ptr
      obj <- (#peek ccl_tree, object) ptr
      d <- (#peek ccl_tree, del) ptr
      nxt <- (#peek ccl_tree, next) ptr
      chds <- (#peek ccl_tree, childs) ptr
      return CCL_TREE {refcount = refc, object = obj, del =d,
                       next =nxt, childs=chds}
  poke ptr (CCL_TREE refcount object del next childs) = do
      (#poke ccl_tree, refcount) ptr refcount
      (#poke ccl_tree, object) ptr object
      (#poke ccl_tree, del) ptr del
      (#poke ccl_tree, next) ptr next
      (#poke ccl_tree, childs) ptr childs
```

Ensuite on récupère la valeur du `CCL_TREE_ST` avec la fonction *peek* et son champ Object avec la méthode *treeObject*.

```
treeObject :: CCL_TREE_ST -> Ptr ()
treeObject (CCL_TREE _ object _ _ _ ) = object
```

Puisque la valeur retournée est un `Ptr ()`, on a besoin de transformer cette valeur en `Ptr ARMOISE_NORMALIZED_PREDICATE`. Ce que l'on fait avec la fonction

```
castPtr :: Ptr a -> Ptr b
```

Après avoir récupéré tous les arguments de la fonction, on peut appeler la fonction `c_alambic_compute_predicate`. On utilise le pointeur null :nullPtr pour tester le résultat retourné depuis C qui est pointeur vers le type abstrait `GENEPI_SET`. S'il y a une erreur, on déréférence le pointeur d'erreur `ALAMBIC_ERROR` qui retourne un entier. Le résultat final sera la valeur de l'erreur retournée `Left alam`.

Si l'appel de la fonction `c_alambic_compute_predicate` est réussi, nous allouons de l'espace à un nouveau pointeur géré avec la fonction C en utilisant un `ForeignPtr`. La valeur spéciale `FinalizerFree` réalise la même opération que la fonction C `free`. Elle désalloue la mémoire qui était allouée avec `ForeignPtr`. Ensuite la fonction retourne le résultat enveloppé dans le type abstrait Solution. Ce résultat est utilisé par la fonction `genepi_set_display_all_solutions` qui le retourne à l'utilisateur. Cette fonction se trouve dans le fichier `genepi.h`. On l'importe dans Haskell en utilisant les mêmes mots clés :

```
foreign import ccall unsafe "genepi.h genepi_set_display_all_solutions"
     c_genepi_set_display_all_solutions :: Ptr GENEPI_SOLVER
                                -> Ptr GENEPI_SET
                                -> Ptr CFile
                                -> Ptr CString
                                -> IO ()
```

Elle prend en entrée un pointeur vers le solveur utilisé pour résoudre la formule de Presburger, le pointeur retourné par la fonction `compute_predicate`, qui représente la solution que l'on souhaite afficher, un pointeur vers une sortie pour l'affichage dans un écran ou l'écriture dans un fichier, et enfin un pointeur vers une chaîne de caractère pour les noms de variables à afficher dans la solution.

Voici le corps de cette fonction :

```
genepi_set_display_all_solutions (Solver solver _) (Sol solution _) =
      withForeignPtr solver $  \ solv -> do
        withForeignPtr solution $  \ sol -> do
                w <- genepi_set_get_width solv sol
                let names = ["x"++ show i | i <- [0..w-1]]
                varnames <- mallocList names
                out <- handleToCFile stdout "w"
                if out == nullPtr
                  then do
                      System.IO.putStrLn "erreur ouverture fic"
                  else do
                      System.IO.putStrLn "fic ouvert"
                      gen\_sol <- c_genepi_set_display_all_solutions
                                  solv sol out varnames
```

```
System.IO.putStrLn "pass"
genepi_loader_terminate
armoise_terminate
ccl_terminate
fflush out
fclose out
```

Voici ce que l'on obtient après exécution de cette fonction :

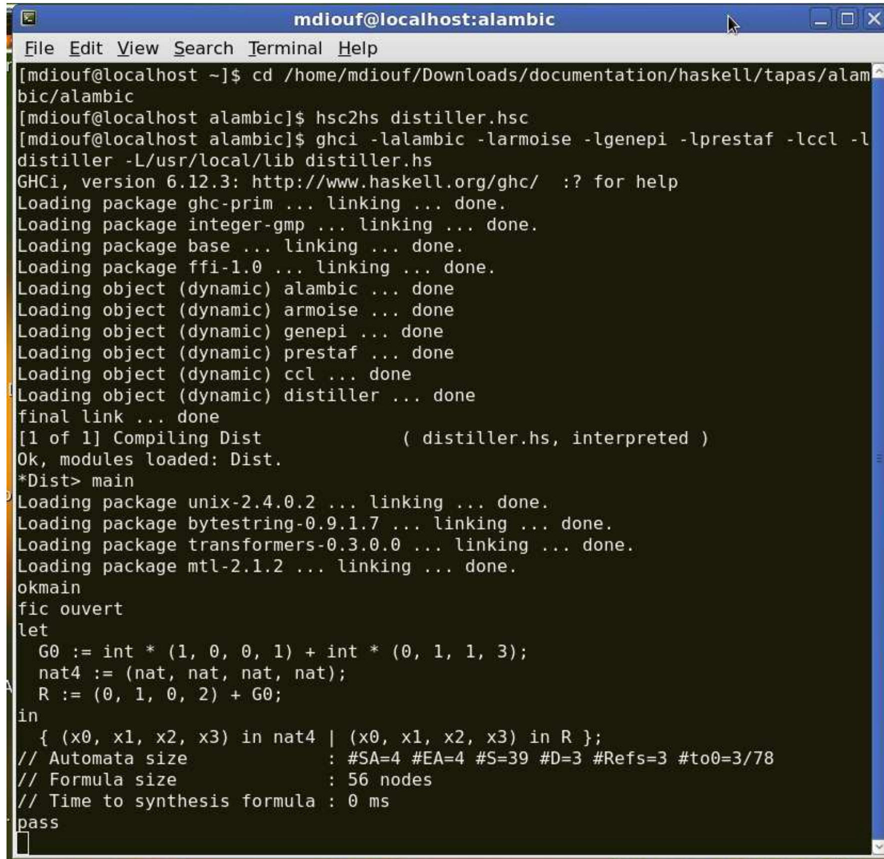


Fig. 4.3 – Résultat de l'exécution de TaPAS dans Haskell

4.3 – Vérification de l'opacité en Haskell

Dans cette troisième et dernière partie de ce chapitre, on utilise les résultats des parties précédentes pour la génération des arbres modulo commutation. On

présente l'implémentation des grammaires vues au chapitre 1 dans le langage fonctionnel Haskell [4]. Nous commençons par quelques outils permettant de manipuler automates d'arbres.

4.3.1 – Quelques outils pour les automates d'arbres

Plusieurs implémentations des automates d'arbres ont été proposées dans la littérature et dans différents langages de programmation. Parmi ces implémentations, on cite celles développées en langage fonctionnel, comme *Timbuk* [5], développé en *OCaml* qui permet de manipuler des automates d'arbres. Ces automates d'arbres y sont implémentés sous forme de tuple de listes, c'est -à-dire ils contiennent une liste de symboles (alphabets), une liste d'opérateurs, une liste d'états, une liste d'états finaux, une liste de transitions et une liste de transitions prioritaires. *Timbuk* supporte toutes les opérations standards sur les automates (union, intersection,...).

Xduce développé par *Hosoya et al*[HVP05] propose un type pour les expressions régulières comme pour la plupart des langages de schémas *XML* avec les notations étoile (*), le choix (|),etc. avec la particularité d'une présentation sémantique de la notion de sous-typage comme une inclusion entre les ensembles de documents identifiés par des types. Voir [HVP05] pour plus de détails.

D'autres outils tels que *MONA*[6] utilisent les automates d'arbres pour calculer des formules logiques. Lethal[7] développé en *Java*, supporte presque toutes les opérations sur les automates d'arbres (déterminisation, complétude,..) et sur les langages (union, intersection,...). *Lethal* implémente aussi les transducteurs d'arbres et les *Hedge automata* et il est très facile d'utilisation. Un autre outil développé en *java* est *TAJA*[8] (*Tree automata for Java*) qui Il propose une implémentation des automates d'arbres et des tuples d'automates d'arbres, ainsi que Il*VATA* [9].

4.3.2 – Expressions régulières et grammaires étendues d'arbres en Haskell

On commence par donner un codage des expressions régulières, qui apparaissent en partie droites de nos grammaires. Ce codage est disponible dans la littérature d'Haskell. On n'a repris quelques fonctions qui nous seront utiles dans

[4]//www.haskell.org
[5]Timbuk : http ://www.irisa.fr/celtique/genet/timbuk/
[6]http ://www.brics.dk/mona/
[7]http ://lethal.sourceforge.net/
[8]www.univ-orleans.fr/lifo/Members/rety/logiciels/taja.php
[9]//www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

notre implémentation présentée dans la section suivante. On représente une expression régulière par la structure de données suivante en Haskell :

```
data Re = Lit Char       -- ^ literal 'a'
        | Epsilon        -- ^ epsilon
        | Concat Re Re   -- ^ concatenation
        | Choice  Re Re  -- ^ choice
        | Empty          -- ^ empty language
        | Union Re Re
        | Star Re        -- ^ kleene's star
          deriving (Eq, Ord, Read)
```

Cette définition correspond à la définition intuitive d'une expression régulière. Par exemple l'expression régulière $A^*B + CD$ s'écrit :

```
Union(Concat ((Star (Lit 'A')) (Lit 'B') Concat(Lit 'C') (Lit 'D'))
```

On peut simplifier cette écriture en définissant une notation infixée pour la concaténation `<>` et l'union `<+>` comme suit :

```
(<>) :: Re -> Re -> Re
Empty <> _ = Empty
_ <> Empty = Empty
Epsilon <> a = a
a <> Epsilon = a
a <> a' = Concat a a'


(<+>) :: Re -> Re -> Re
Empty <+> a = a
a <+> Empty = a
a <+> a' = Union a a'
```

De même pour l'étoile de kleene comme suit :

```
star :: Re -> Re
star Empty    = Epsilon
star Epsilon  = Epsilon
star (Star a) = Star a
star a        = Star a
```

L'opérateur (+) s'en déduit :

```
 A+ =  A <> star A
```

L'expression régulière A^*B+CD peut alors s'écrire `(star A<>B)<+>(C<>D)`. On peut définir d'autres fonctions auxiliaires comme la fonction `fromString` qui permet de convertir une chaine de caractères en expression régulière ou `toString` la fonction inverse ou encore `printRE` qui permet d'afficher une expression régulière.

```
instance IsString Re where
   fromString cs = foldr ((<>) . Lit) Epsilon cs
```

Par exemple

```
fromString "ABC" = Lit 'A' <> Lit 'B' <> Lit 'C' <> Epsilon
```

La fonction *match* définie ci-dessous permet de tester si une chaine correspond à une expression régulière.

```
match :: Re -> String -> Bool
match r cs = accept r cs null
```

Elle se déduit de la fonction *accept* proposée dans [ON01] :

```
accept :: Re -> String -> (String -> Bool) -> Bool
accept Empty cs k   = False
accept Epsilon cs k = k cs
accept (Lit c) (c':cs) k = c==c' && k cs
accept (Lit c) []  k   = False
accept (Concat r1 r2) cs k = accept r1 cs (\cs' -> accept r2 cs' k)
accept (Union r1 r2) cs k = accept r1 cs k || accept r2 cs k
accept (Star r) cs k = accept_star r cs k

accept_star r cs k
  = k cs || accept r cs (\cs' -> cs'/=cs && accept_star r cs' k)
```

L'adaptation de la fonction `match` pour les expressions régulières modulo commutations.

```
acceptP ::  Re -> [String] -> ([String] -> Bool) -> Bool
acceptP Empty css k   = False
acceptP Epsilon css k = k css
acceptP (Lit c) [[]]  k   = False
acceptP (Lit c) (cs:css) k = (accept (Lit c) cs null) || k css
acceptP (Concat r1 r2) css k = acceptP r1 css (\css' -> acceptP r2 css' k)
```

```
acceptP (Union r1 r2) css k = acceptP r1 css k || acceptP r2 css k
acceptP (Star r) css k = accept_starP r css k

accept_starP r css k
  = k css || acceptP r css (\css' -> css'/=css && accept_starP r css' k)
```

La fonction `matchP` qui lui est associée est donnée par :

```
matchP :: Re -> String -> Bool
matchP r cs = acceptP r (permutations cs) null
```

Cette fonction sera utile notamment lorsqu'on voudra vérifier si un arbre est conforme à une grammaire donnée.

A présent on peut introduire les grammaires régulières d'arbres qui utilisent les expressions régulières en la partie droite de leurs règles. Le code Haskell suivant est une extension du code proposé dans [BT08] par des expressions régulières. La grammaire est donnée par :

```
prods   :: Gram prod re -> [prod]
symbols :: Gram prod re -> [symb]
lhs     :: Gram prod re -> prod -> re
rhs :: Gram prod re -> prod -> re

data Gram prod re = Gram {prods :: [prod],
                          symbols :: [re],
                          lhs :: prod -> re,
                          rhs :: prod -> re}
```

On vient de définir la structure de données `Gram prod re`. Elle est composée d'un ensemble de productions `prod` représentant le type des productions et `re` le type des expressions régulières. La variable `prods` retourne la liste des productions, `symbols` donne la liste des symboles grammaticaux. `lhs` et `rhs` donnent respectivement le symbole grammatical se trouvant en partie gauche d'une règle et l'expression régulière en partie droite d'une règle. Essayons de représenter notre système et notre secret décrit par les règles suivantes :

$$\begin{aligned} Gram : \omega 1 &: A \rightarrow C^*B \\ \omega 2 &: A \rightarrow \varepsilon \\ \omega 3 &: B \rightarrow CA \\ \omega 4 &: B \rightarrow BB \\ \omega 5 &: C \rightarrow AC \\ \omega 6 &: C \rightarrow CC \\ \omega 7 &: C \rightarrow \varepsilon \end{aligned}$$

$$\begin{aligned} Sec : \omega 1 &: A \rightarrow BC^* \\ \omega 2 &: A \rightarrow \varepsilon \\ \omega 3 &: B \rightarrow DB \\ \omega 4 &: B \rightarrow BB\varepsilon \\ \omega 9 &: B \rightarrow \varepsilon \\ \omega 5 &: C \rightarrow CA \\ \omega 6 &: C \rightarrow D \\ \omega 8 &: C \rightarrow \varepsilon \\ \omega 7 &: C \rightarrow CCB \end{aligned}$$

Cela donne dans notre structure de données :

```
data Prod = W1 | W2 | W3 | W4 | W5 | W6 | W7 | W8 | W9 deriving (Eq, Show)
gram :: Gram Prod Re
gram = Gram lprod lre lhs_ rhs_
  where lprod = [W1, W2, W3, W4, W5, W6, W7, W8, W9]
        lre = [fromString "A", fromString "B", fromString "C", fromString "D"]
        lhs_ p = case p of
                 W1 -> fromString "A"; W3 -> fromString "B";
                 W5 -> fromString "C"; W7 -> fromString "C"
                 W2 -> fromString "A"; W4 -> fromString "B";
                 W6 -> fromString "C" ; W8 -> fromString "C";
                 W9 -> fromString "B"
        rhs_ p = case p of
                  W1 -> fromString "B" <>  star (fromString "C");
                  W3 -> fromString "D" <> fromString "B";
                  W5 -> fromString "C"<> fromString "A";
                  W7 -> fromString "D";
                  W2 -> Epsilon; W4 -> fromString "B" <> fromString "B";
                  W6 -> fromString "C" <> fromString "C" <> fromString "B";
                  W8 -> Epsilon; W9 -> Epsilon

sec :: Gram Prod Re
sec = Gram lprod lre lhs_ rhs_
  where lprod = [W1, W2, W3, W4, W5, W6, W7]
        lre = [fromString "A", fromString "B", fromString "C", fromString "D"]
        lhs_ p = case p of
                 W1 -> fromString "A"; W3 -> fromString "B";
                 W5 -> fromString "C";W7 -> fromString "C" ;
```

```
                W9 -> fromString "B"
                W2 -> fromString "A"; W4 -> fromString "B";
                W6 -> fromString "C" ; W8 -> fromString "D"
       rhs_ p = case p of
                 W1 -> fromString "B" <> fromString "C";
                 W3 -> fromString "B" <> fromString "D";
                 W5 -> fromString "C"<> fromString "A";
                 W7 -> fromString "D"
                 W2 -> Epsilon; W4 -> Epsilon;
                 W6 -> fromString "C" <> fromString "C" <> fromString "B"
```

pour tester ce programme avec Haskell, il nous faut installer GHC[?] ensuite tester en ligne de commande en entrant la commande suivante par exemple :

```
mdiouf@localhost:ExempleHaskell
File Edit View Search Terminal Help
[mdiouf@localhost ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> symbols gram
[A,B,C]
*RE> prods gram
[W1,W2,W3,W4,W5,W6,W7]
*RE> rhs gram W1
(B(C)*)
*RE> lhs gram W1
A
*RE>
```

FIG. 4.4 – Grammaire d’arbres en Haskell

Les arbres décrits au chapitre 1 sont représentés par la structure de données suivante :

```
data Tree a = Node{top::a, succ_ ::[Tree a]} deriving (Eq,Show)
```

Un exemple d’un tel arbre est :

```
tree1= Node {top = ’A’,succ_= [Node ’C’ [],Node ’B’ [Node ’C’ [], Node ’A’ []]]}
```

Pour vérifier que cet arbre est conforme à la grammaire décrite ci-dessus, on donne une adaption de la fonction `isProd` et `der2ast` tirée de [BT08] par extension aux expressions régulières. Ces deux fonctions vérifient la conformité d'un arbre.

```
isProd ::  Gram prod Re -> Char -> String -> Maybe prod
isProd gram symb symbs =
    case [p | p <- prods gram, match (lhs gram p) (symb:[]),
                      match (rhs gram p) symbs] of
         [] -> Nothing
         [p] -> Just p
         otherwise -> error "two productions with identical left
                                  and right hand sides"
```

La fonction `isProd` ci-dessous permet de rechercher une production donnée par ses parties droite et gauche. Elle prend en entrée une grammaire, un symbole et une suite de symboles et teste si c'est une production de la grammaire. Pour le tester il suffit de saisir la ligne suivante : cette fonction est définie pour être utilisée dans la fonction qui suit, qui teste si un arbre dont les noeuds sont étiquetés par des symboles grammaticaux est conforme à la grammaire et retourne alors les règles qui lui sont associées :

```
der2ast ::  Gram prod Re -> Tree Char -> Maybe (Tree prod)
der2ast gram (Node symb ts) =
 do
          p <- isProd gram symb (map top ts)
          ts' <- mapM (der2ast gram) ts
          return (Node p ts' )
```

Le premier `der2ast` donne un résultat positif. L'arbre est reconnu conforme à la grammaire `gram`. Alors que le deuxième `der2ast` teste la reconnaissance de l'arbre généré par la règle $A \rightarrow BC^*$, sans l'apparition de la sorte C (étoile de Kleene).

Il nous reste à définir la projection. On code d'abord le théorème de *Parikh* qu'on rappelle dans la section suivante. Ce qui nous permet de déterminer la grammaire projetée.

Théorème 4.3. *[Reu89] Si la règle pour un symbole A se factorise (modulo commmutation) en $A \rightarrow Q(A).A{+}T$, où A n'apparaît pas dans T, alors $L(G, A) = Q(T)^*T$ est la plus petite solution de l'équation associée à cette grammaire.*

```
mdiouf@dhcp-172-23-49-236:ExempleHaskell
File Edit View Search Terminal Help
[mdiouf@dhcp-172-23-49-236 ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> isProd gram 'C' "CA"
Just W5
*RE> isProd gram 'B' "DB"
Just W3
*RE> isProd gram 'B' ""
Just W9
*RE> isProd gram 'B' "CA"
Nothing
*RE> der2ast gram Node {top = 'A',succ_= [Node 'B' [],Node 'C' [Node 'C' [], Nod
e 'A' []]]}
Just (Node {top = W1, succ_ = [Node {top = W9, succ_ = []},Node {top = W5, succ_
 = [Node {top = W8, succ_ = []},Node {top = W2, succ_ = []}]}]})
*RE> der2ast gram Node {top = 'A',succ_= [Node 'B' [Node 'B' [], Node 'B' []]]}
Just (Node {top = W1, succ_ = [Node {top = W4, succ_ = [Node {top = W9, succ_ =
[]},Node {top = W9, succ_ = []}]}]})
*RE>
```

FIG. 4.5 – Conformité d'un arbre à la grammaire d'arbres *gram*

Pour une règle donnée, la solution associée à un symbole symb est :

```
solution :: Gram prod Re -> Re -> Re
solution gram symb =
    foldr1 (<>) ([star (foldl1 (<+>) (init ts2))] ++ (last ts2:[]))
                   where
           ts2 = [foldr ((<>) . Lit) Epsilon x | x <- ts1]
           ts1 = [replace (toString symb) (last ts1) xs | xs <-ts ]
           ts = [delete (head (toString (symb))) xs | xs <-tss ]
           tss = [toString x | x <- (rule gram symb) ]
```

Dans cette fonction `solution` on trouve la fonction `rule` :

```
rule :: (Eq Re) => Gram prod Re -> Re -> [Re]
rule gram symb =
   [rhs gram p | p <- prods gram, symb == lhs gram p]
```

Elle prend en entrée une grammaire et une sorte définie dans la grammaire, représentant la partie gauche d'une règle et retourne la partie droite. Elle est similaire à la fonction `rhs` qui au lieu de prendre une sorte prend un symbole.

Les fonctions `foldr` (respectivement `foldl`) sont des combinateurs Haskell permettant d'itérer un opérateur binaire sur une liste non vide d'arguments en l'associant à droite (respectivement à gauche). L'exécution de cette fonction sur

la sorte C non visible donne :

```
mdiouf@localhost:ExempleHaskell
File Edit View Search Terminal Help
[mdiouf@localhost ~]$ cd /home/mdiouf/Downloads/documentation/haskell/ExempleHas
kell/
[mdiouf@localhost ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )

regularExpression.hs:219:12: Not a data constructor: `w1'
Failed, modules loaded: none.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> solution gram (Lit 'C')
(((A+(DB)))*D)
*RE> rule gram (Lit 'C')
[(CA),((CC)B),D]
*RE>
```

Fig. 4.6 – Théorème de Parikh en Haskell

Pour déterminer la grammaire projetée, on introduit la fonction `mapReg` qui permet d'appliquer une fonction à chaque élément d'une liste d'expressions régulières :

```
mapReg :: ([Re] -> [Re]) -> [Re] -> [Re]

mapReg f [Epsilon] =  f [Epsilon]
mapReg f [Lit c] = f [Lit c]
mapReg f [Star r] = [Star (head (mapReg f [r]))]
mapReg f [Union r1 r2] = [Union (head (mapReg f [r1])) (head (mapReg f [r2]))]
mapReg f [Concat r1 r2] = [Concat (head (mapReg f [r1])) (head (mapReg f [r2]))]
mapReg f (x:xs) =mapReg f [x] ++ mapReg f xs
```

A présent la projection d'une grammaire suivant une sorte non visible par rapport à une règle est donnée par la fonction `proj` :

```
proj :: Gram prod Re -> Re -> [Re] -> [Re]
proj gram symb rs = mapReg (replace (symb:[]) (solution gram (symb):[])) rs
```

La fonction `proj` ne fait que remplacer la solution trouvée par la fonction `solution` dans les parties droites des règles contenant la ou les sortes non visibles en question. Elle prend en entrée une grammaire, la sorte non visible et la partie droite

d'une règle.

```
[mdiouf@localhost ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> rule gram (Lit 'A')
[((C)*B),€]
*RE> proj gram (Lit 'C') (rule gram (Lit 'A'))
[(((((A+(DB)))*D))*B),€]
*RE>
```

FIG. 4.7 – Partie droite de la règle A dans la grammaire projetée

Les arbres projetés sont déterminés grâce à la notion de vue associée à des sortes. On retrouve cette fonction dans [BT08] :

```
viewAB :: Char -> Bool
viewAB symb = case symb of
                 'A' -> True
                 'B' -> True
                 'C' -> False
                 'D' -> False
```

qui donne une vue suivant les sortes A et B et qui efface les sortes C et D non visibles. Cette fonction est définie comme un prédicat sur les sortes. La fonction `viewAC` est duale à la fonction `viewAB`. Elle donne une vue suivant les sortes C et D et efface les sortes A et B.

```
viewCD :: Char -> Bool
viewCD symb = case symb of
                'A' -> False
                'B' -> False
```

```
                'C' -> True
                'D' -> True
```

La fonction projection associée à ces vues :

```
projection :: (Char -> Bool) -> Tree Char -> [Tree Char]
projection view der = if view (top der) then [Node (top der) sons] else sons
  where sons = concat (map (projection view) (succ_ der))
```

Cette fonction prend une vue et un arbre et retourne un arbre dans lequel les sortes non visibles sont effacées.
Les projections suivants les sortes A, B et C et D sont données par les fonctions :

```
projAB = projection viewAB
projAC = projection viewCD
```

Il ne reste plus qu'à le tester.

```
mdiouf@localhost:ExempleHaskell
File Edit View Search Terminal Help
[mdiouf@localhost ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> projAB Node {top = 'A',succ_= [Node 'B' [],Node 'C' []]}
[Node {top = 'A', succ_ = [Node {top = 'B', succ_ = []}]}]
*RE> projAB Node {top = 'A',succ_= [Node 'C' [],Node 'B' [Node 'C' [], Node 'A'
[]]]}
[Node {top = 'A', succ_ = [Node {top = 'B', succ_ = [Node {top = 'A', succ_ = []
}]}]}]
*RE> 
```

FIG. 4.8 – Projection d'arbre de la grammaire suivant les sortes visibles A et B

On rappelle que l'opacité est donnée par $p_{\Xi'}(S \cap G) \setminus p_{\Xi'}(G \setminus S)$. Dans cette formule on retrouve la fonction de projection, l'intersection et la différence. Il nous reste les fonctions qui calculent l'intersection et la différence. Les fonctions `inter` et `diff` suivantes permettent de calculer l'intersection et la différence entre deux expressions régulières. Quant aux fonctions `int` et `dif`, elles calculent la différence et l'intersection de liste d'expressions régulières.

```
-- intersection de 2 expressions régulières
inter ::Re -> Re -> Re
inter Epsilon _ = Epsilon
inter _  Epsilon = Epsilon
inter (Lit a) (Lit a') = if ((Lit a) == (Lit a')) then Lit a else Epsilon

inter (Star a) (Star a') = Epsilon
inter (Star a) a' = inter a a'

inter (Union a a') (Union b b') = inter a b <> inter a' b'<> inter a b'
                                            <> inter a' b
inter (Union a a') (Concat b b')  = inter a b <> inter a' b'
inter (Union a a') b  = inter a b <> inter a' b --(a' <^> b)
```

```
inter (Concat a a’) (Concat b b’)  = inter a b <> inter a’ b’
inter (Concat a a’) (Union b b’) = inter a b <> inter a’ b’
inter (Concat a a’)  b  = inter a b <> inter a’ b

int:: Eq Re => [Re] ->[Re] ->[Re]
int [] [] = []
int [] xs = []
int xs [] = []
int (x:xs)(y:ys) = inter x y : int xs ys

--Difference de 2 expressions régulières
diff :: Re  -> Re  -> Re
diff Epsilon _ = Epsilon
diff r Epsilon = r
diff  (Lit a) (Lit a’) =  if ((Lit a) == (Lit a’)) then Epsilon else Lit a

diff (Star a) (Star a’) = Concat a (Star a)
diff (Star a) a’ = diff a a’

diff (Union a a’) (Union b b’) = diff a b <> diff a’ b’<> diff a b’ <> diff a’ b
diff (Union a a’) (Concat b b’)  = diff a b <> diff a’ b’
diff (Union a a’) b  = diff a b <> diff a’ b

diff (Concat a a’) (Concat b b’)  = diff a b <> diff a’ b’
diff (Concat a a’) (Union b b’) = diff a b <> diff a’ b’
diff (Concat a a’) b = diff a b <> diff a’ b


dif :: [Re] -> [Re] -> [Re]
dif [] [] = []
dif [] ys = []
dif xs [] = xs
dif (x:xs) (y:ys) = diff x y : dif xs ys
```

Ayant obtenu ces fonctions, le calcul de l’intersection et la différence de deux grammaires d’arbres devient facile. Il est obtenu par :

```
-- intersection de deux automates
(<£>)     :: (Eq a) => [(a, [Re])] -> [(a, [Re])] -> [Re]
gram <£> sec = [res|
```

```
                  (a1, r1) <- gram,
                  (a2, r2) <- sec,
                  a1 == a2,
                  res <- int r1 r2]

-- différence de deux automates
(<$>)     :: (Eq a) => [(a, [Re])] -> [(a, [Re])] -> [Re]
gram <$> sec = [res|
                  (a1, r1) <- gram,
                  (a2, r2) <- sec,
                  a1 == a2,
                  res <- dif r1 r2]
```

A partir de là, on peut vérifier l'opacité par le code suivant :

```
verifOpa:: [Re] -> [Re] -> Re -> [Re]
verifOpa rs1 rs2 symb = dif (proj gram symb rs1) (proj gram symb rs2)
```

Cette fonction prend en entrée deux listes `rs1` et `rs2` représentant respectivement le résultat du calcul de l'intersection et de la différence de nos grammaires `gram` et `sec`. `Symb` représente la sorte non visible suivant laquelle on effectue la projection.

```
mdiouf@localhost:ExempleHaskell
File Edit View Search Terminal Help
[mdiouf@localhost ~]$ cd /home/mdiouf/Downloads/documentation/haskell/ExempleHas
kell/
[mdiouf@localhost ExempleHaskell]$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load regularExpression
[1 of 1] Compiling RE               ( regularExpression.hs, interpreted )
Ok, modules loaded: RE.
*RE> let ts1 = gram2coalg gram (Lit 'A')
*RE> let ts2 = gram2coalg sec (Lit 'A')
*RE> verifOpa (ts1 <£> ts2) (ts1 <$> ts2) (Lit 'C')
[(B(((A+(DB)))*D)),€]
*RE> ts1 <£> ts2
[(BC),€]
*RE> ts1 <$> ts2
[€,€]
*RE> ts1
[(W1,[(B(C)*)]),(W2,[€])]
*RE> ts2
[(W1,[(BC)]),(W2,[€])]
*RE>
```

Fig. 4.9 – Vérification de l'opacité

La figure 4 correspond à un système non opaque. En effet un système opaque retournerait une liste vide. Le résultat de la fonction `verifOpa` affiche un contre exemple.

Conclusion

Dans cette thèse nous avons proposé une formalisation de l'opacité des artefacts dans les systèmes workflow centrés sur l'échange de données structurées. Nous avons identifié deux contextes dans lesquels non seulement l'opacité est décidable mais l'algorithme de décision peut générer des contre exemples à l'opacité lorsque cette propriété n'est pas satisfaite : *(i)* lorsque les grammaires sont non récursives ou *(ii)* lorsque les documents ne sont considérés qu'à la permutation près de sous arbres.

Le retour d'information produit par l'algorithme de décision peut être utilisé par le concepteur du système pour opérer les modifications nécessaires au rétablissement de l'opacité. On peut souhaiter que le système aille un peu plus loin en effectuant lui-même ces modifications ou tout du moins en proposant à l'utilisateur des modifications de nature à résoudre les éventuels problèmes de fuite d'information.

De telles techniques, dite de contrôle, ont été étudiées dans le contexte des systèmes à événement discrets, cadre dans lequel l'opacité a été étudiée jusque là. Rappelons brièvement ce qui a été fait pour assurer l'opacité dans les systèmes à événement discrets. Dans ce cas l'univers U est donné par un langage régulier dont les éléments (des mots) sont toutes les exécutions du système, c'est-à-dire les suites d'événements (actions internes ou interactions avec un utilisateur) qui peuvent advenir dans le système. Chaque observateur ne voit que les actions avec lesquelles il interagit avec le système (un sous alphabet de l'ensemble E des événements). La fonction d'observation qui lui est associée $\phi : U \to O$ est la projection qui efface dans toutes les exécutions toutes les occurrences d'événements non observables. Comme *(i)* l'ensemble des langages réguliers est clos de manière effective par les opérations booléennes, *(ii)* l'image d'un langage régulier par un morphisme de monoïdes est (de manière effective) un langage régulier, *(iii)* on peut décider de la vacuité d'un langage régulier (et donc aussi de l'inclusion de deux langages réguliers) on en déduit qu'on peut décider si $\phi(S) \subseteq \phi(U \setminus S)$, c'est-à-dire de l'opacité de S.

Le langage régulier $L_{\varphi,S} = \phi^{-1}\left(\phi(S) \setminus \phi(U \setminus S)\right)$ donne toutes les exécutions qui témoignent de la non-opacité de P vis-à-vis du secret S. De façon plus précise le langage $L = U \setminus (\bigcup \{\phi^{-1}\left(\phi(S) \setminus \phi(U \setminus S)\right) \mid S \text{ secret pour } \phi\})$ est le plus grand

langage opaque contenu dans U. Si on fait l'hypothèse que toutes les actions sont contrôlables on peut rendre le système opaque en le synchronisant avec un contrôleur donné par le langage précédant. C'est-à-dire qu'on autorise le système à exécuter une séquence que lorsque celle-ci est de façon simultanée autorisée, c'est-à-dire exécutable, par le contrôleur. Le langage du système contrôlé est donc l'intersection du langage du système d'origine avec celui du contrôleur. Ce contrôleur est optimal en ce sens qu'il n'interdit que les comportements qui violent l'opacité du système. Qui plus est [BBB+07] ce contrôle est distribué, c'est-à-dire qu'on peut de façon équivalente associer à chaque utilisateur ϕ un contrôleur qui restreint les interactions autorisées de cet utilisateur avec le système. Des résultats analogues existent également sous des hypothèses plus faibles, c'est-à-dire lorsqu'on ne suppose plus que toutes les actions puissent être contrôlées.

Nous disposons ici de la même caractérisation logique du contrôleur quitte à remplacer des automates de mots par des automates d'arbres (non ordonnés et à arité variables). Par ailleurs nous savons [BT08] que la synchronisation de vues partielles d'un même document peut s'obtenir via le produit des automates d'arbres associés à chacune de ces vues. Les techniques de synthèse et de distribution d'un contrôleur devraient pouvoir s'étendre sans difficultés majeures aux artefacts d'un workflow.

La solution obtenue par contrôle produit une grammaire qui génère exactement les documents de U pour lesquels aucun secret n'est dévoilé. Le résultat peut néanmoins être peu satisfaisant d'un point de vue pratique si cela force à exclure trop de documents, et en particulier des documents que le concepteur du système souhaiterait conserver dans son modèle. Une solution moins intrusive, qui s'inspire de techniques d'extensions conservatives de schémas XML, est à l'étude. Cette méthode consiste à n'invalider aucun document mais au contraire à étendre la grammaire : l'introduction de nouveaux documents licites constitue du “bruit” qui peut empêcher la détection de secrets.

Une suite de ce travail va s'attacher à préciser la forme des règles sémantiques qui régissent le calcul des attributs aux noeuds des documents. Cela déterminera par la même occasion la syntaxe pour les algèbres et donc en particulier pour les requêtes faites sur un document et pour l'expression des secrets. Nous comptons également implémenter les algorithmes de décision et de contrôle de l'opacité en utilisant les bibliothèques existantes de manipulation des ensembles semi-linéaires.

Annexe
Arithmétique de Presburger

Dans cette annexe nous détaillons les passages entre ensembles semi-linéaires, formules de Presburger et automates. Rappelons que les parties rationnelles de N^k coïncident avec les parties semi-linéaires définies comme suit.

Définition 4.4. *Une partie linéaire de N^k est une partie de la forme*

$$v_0 + n_1 \cdot v_1 + \cdots + n_\ell \cdot v_\ell$$

C'est une expression affine dans laquelle le vecteur v_0 est l'origine et les vecteurs $v_1, \ldots v_\ell$ sont les vecteurs de base fixés. Les variables n_i représentent des entiers arbitraires. Une partie semi-linéaire de N^k est une union finie de parties linéaires de N^k, donc s'écrit de la forme

$$\cup_n (v_0 + n_1 \cdot v_1 + \cdots + n_\ell \cdot v_\ell)$$

Théorème 4.5. *[GS66b] La famille des parties rationnelles de N^k est close par union, intersection et complémentation. La projection d'un ensemble semilinéaire est semilinéaire.*

Définition 4.6. *Les formules de Presburger est le plus petit ensemble contenant les formules $n_0 + \sum_1^m n_i x_i = n'_0 + \sum_1^m n'_i x_i$ dans lesquelles les n_i et n'_i sont des constantes entières et les x_i sont des variables, clos par les opérations booléennes et par quantification.*

Exemple 4.7. $\exists\, (x_1, x_2, y_1, y_2) \in N^4$ *tel que* $-x_1 + x_2 - y_2 = 3$ *est une formule de Presburger.*

Théorème 4.8. *[GS66b] Les ensembles de Presburger sont stables par intersection, par union, par somme, par passage au complémentaire, par morphisme de monoïdes et par morphisme inverse.*

Théorème 4.9. *[GS66b] La famille des ensembles de Presburger de N^n est identique à la famille des ensembles semi-linéaires de N^n. De plus chaque description est effectivement calculable à partir de l'autre.*

Exemple 4.10. *Soit* $\{(x, y) \in N^2 | x \geq y\}$ *un ensemble linéaire. Alors on peut le représenter par la formule de Presburger suivante :* $F(x, y) \equiv (\exists z) \cdot x = y + z$

L'intérêt de la logique de Presburger est sa décidabilité (au contraire de l'arithmétique dans son ensemble, qui est indécidable). Comme pour la décision de la logique du premier ordre, la méthode consiste à construire (par induction sur la structure de la formule) un automate qui reconnait l'ensemble des vecteurs vérifiant la formule pour un codage particulier des vecteurs en des mots sur un certain alphabet que nous décrivons dans cette annexe. La validité de la formule se réduit à la non vacuité de l'automate correspondant.

Définition 4.11. *Soit* A *un automate définit par* $A = (Q, E, q_0, \Delta, F)$*, avec*

- Q *: ensemble des états ;*
- *L'alphabet* E *des lettres.*
- *l'état initial* $q_0 \in Q$
- *l'ensemble des états finaux* $F \subseteq Q$*.*
- *La fonction de transition* Δ *définie telle que* $\Delta \subseteq Q \times E \times Q$*.*

t *est une transition ou un arc de* Δ*, s'il existe* q*,* q' *appartenant à* Q*, et* e *dans* E *telle que* $t = (q, e, q')$*. On pose alors* $l(t) = e$*,* $\partial_0(t) = q$*,* $\partial_1(t) = q'$*.*

Soit le graphe orienté Δ *associé à l'automate. Dans la suite, on essaie de caractériser les vecteurs qui sont l'image commutative du graphe* Δ*. On traduit cette caractérisation par une formule de Presburger.*

Un chemin dans le graphe Δ est une suite finie d'arc noté c et défini par $c = (t_i)_{1 \leq i \leq n}$ avec la condition $\partial_1(t_{i-1}) = \partial_0(t_i)$ pour tout i, $1 \leq i \leq n$. On peut alors poser $\partial_0(c) = \partial_0(t_1), \partial_1(c) = \partial_1(t_n)$.

Définition 4.12. *Le graphe* Δ *est dit connexe si pour tout couple de sommets* s *et* t *il existe un chemin de* s *à* t*. On peut associer à chaque chemin* c *le vecteur* $\bar{c} = \sum_{t \in \Delta} \bar{c}(t).t \in N^{\Delta}$ *qui compte le nombre d'occurences de l'arc* t *dans le chemin* c*.*

Le bilan $\partial(c) = \partial_1(c) - \partial_0(c)$ d'un chemin c détermine son origine et son extrémité. La remarque suivante montre que le bilan d'un chemin ne dépend que de son image commutative

Remarque. — $\partial(c) = \sum_{t \in \Delta} \pi(\bar{c}(t)).\partial(t)$

Soit c un chemin acceptant de A tel que $v = l(\bar{c})$ compte le nombre d'occurrences des lettres qui apparaissent dans le chemin c. Le lemme suivant donne une caractérisation des vecteurs qui sont images commutatives d'un chemin :

Lemme 4.13. *[Reu89] Soit $\pi = \bar{c}$. Alors π est l'image commutative d'un chemin de Δ si et seulement si les deux conditions suivantes sont vérifiées :*

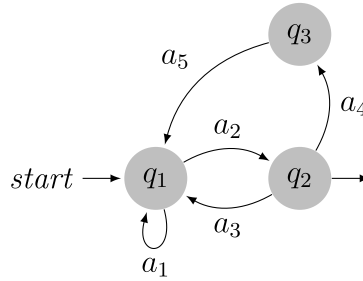
1. *Le graphe restreint à π Δ_π est connexe.*
2. *le bilan de π, $\partial(\pi)$ est la différence de deux états.*

Soit $v = (y_1, \ldots, y_n)$ avec $n = |E|$. La caratérisation énoncée dans le lemme précédent s'exprime par la formule de Presburger :

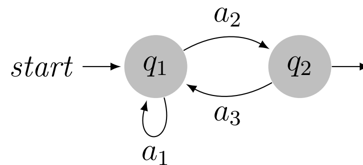
$$\begin{array}{lll} \Psi(v) & \equiv & \exists \pi = (x_1, \ldots, x_n). \\ & & (\vee_{q \in F} \vee_{\Delta' \subseteq \Delta_q} (\partial(\pi) = q - q_0 \wedge supp(\pi) = \Delta') \wedge (v = \sum x_i l(t_i))) \end{array}$$

dans laquelle *supp* est le support de Δ'.

Exemple 4.14. *Soient G le graphe suivant*

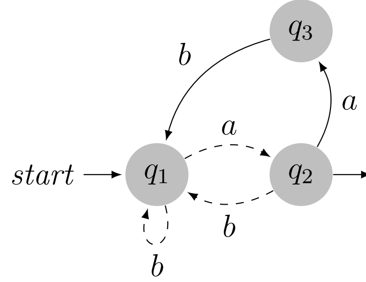


et c le chemin $a_1a_1a_1a_2a_3a_2a_3a_2$, le bilan de c est $\partial(c) = t - s$ et son image commutative est $\overline{c} = 3a_1 + 3a_2 + 2a_3$. Avec nos conventions, $\overline{c}$ sera aussi représenté par le vecteur $\overline{c} = (3, 3, 2, 0, 0)$. Si c, c' sont deux chemins consécutifs, alors on a $\partial(cc') = \partial(c) + \partial(c')$ et $\overline{cc'} = \overline{c} + \overline{c'}$. En particulier, si $c = a_1 \ldots a_n$ $(a_i \in A)$ alors $\partial(c) = \sum_{i=1}^{n} \partial(a_i) = \sum_{a \in A} |c|_a \partial(a)$. Ceci montre que le bilan d'un chemin ne dépend que de son image commutative. A quelle condition un élément de N^5 est-il image commutative d'un chemin ? Soit $v = 3a_1 + 3a_2 + 2a_3$. On a $\partial(v) = t - s$. De plus $G|v$ est le graphe

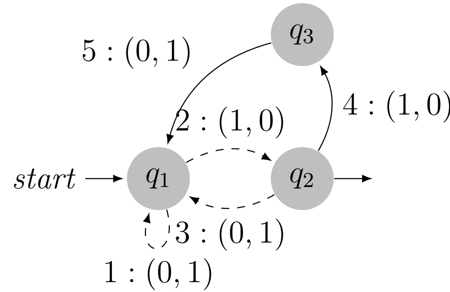


Comme on l'a vu, v est l'image commutative du chemin $a_1a_1a_1a_2a_3a_2a_3a_2$, mais aussi du chemin $a_1a_2a_3a_1a_2a_3a_1a_2$. Si v est l'image commutative d'un chemin, alors $G|v$ est un graphe connexe ; de plus le bilan de v est de la forme $t - s$ pour deux sommets s, t du graphe (éventuellement égaux). On va voir que ces deux conditions caractérisent les images commutatives de chemins acceptants.

Exemple 4.15. *Soit un automate $A = (\Sigma, Q, q_1, \delta, q_3)$ sur A^*, avec comme alphabet $\Sigma = \{a, b\}$, q_1 est l'état initial, q_3 est l'état final. les traits pleins représentent les chemins acceptants.*



On remplace les étiquettes par des vecteurs sur $N^{|\Sigma|}$, i.e N^2 :



On montre d'abord que la conditon 2 du lemme traduite par cette formule $\phi(v) = \exists \pi = (x_1, \ldots, x_n) \in N^{\Delta} \quad (\vee_{q \in F} \partial(\pi) = q - q_0) \wedge (v = \sum_{1 \leq i \leq m} x_i.l(t_i))$ est une condition nécessaire, où les x_i sont les arcs et les i les numéros d'arcs comme indiqués sur la figure ci-dessus et π un chemin acceptant de l'automate. Cette formule $\phi(v)$ dit tout simplement que le bilan de π est la différence de deux états et v est bien le vecteur correspondant au chemin π .

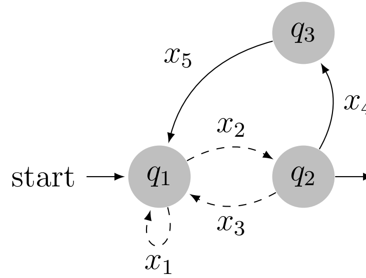
Soit $v = (y_1, y_2)$ avec $2 = |E|$, v $= \sum_{1 \leq i \leq m} x_i.l(t_i)) = l(\pi)$ le vecteur correspondant au chemin π. La formule précédente s'écrit $\phi(v) = \phi(y_1, y_2) = \exists \pi = (x_1, x_2, x_3, x_4, x_5) \in N^4$. On peut alors calculer le bilan de chaque chemin :

$$\begin{aligned}
\partial(x_1) &= q_1 - q_1 = 0. \\
\partial(x_2) &= q_2 - q_1. \\
\partial(x_3) &= q_1 - q_2 \\
\partial(x_4) &= q_3 - q_2 \\
\partial(x_5) &= q_1 - q_3
\end{aligned}$$

Ce bilan doit être égal à $\partial(\pi) = q_2 - q_1$. Ce qui amène à poser le système d'équations suivant :

$$\begin{cases} x_3 - x_2 = -1 \\ x_2 - x_3 = 1 \\ x_4 = x_1 + 2 * x_2 - x_3 \end{cases}$$

On a pas pris en compte dans ce système les arcs menant ou quittant l'état q_3. On s'est restreint au sous graphe G—v. Si $\phi(v)$ est vraie on a $\partial(\pi) = q - q_0$ pour $q \in F$. Le point 2. du Lemme est satisfait. Si Δ_π est connexe alors $\pi \in N^\Delta$ sera l'image commutative d'un chemin acceptant et puisque $v = l(\pi)$



Le support intervenant dans la formule de Presburger $\Psi(v)$ est donné par :

$$supp(\pi) = \Delta' \Leftrightarrow \begin{cases} x_2 \geq 1 \\ x_1 \geq 0 \\ x_3 \geq 0 \\ x_4 = 0, x_5 = 0 \end{cases}$$

Le support $supp(\pi)$ indique que Δ_π est connexe. Comme on le montre dans la figure ci-dessous , seuls les traits pleins doivent être empruntés. Donc la formule suivante $\exists\ \pi = (x_1, x_2, x_3, x_4) \in N^4$

$$\Psi(y_1, y_2) = \Delta' \Leftrightarrow \begin{cases} x_2 \geq 1 \\ x_3 - x_2 = -1 \\ x_2 - x_3 = 1 \\ x_4 = x_1 + 2 * x_2 - x_3 \end{cases}$$

donne bien une traduction du lemme par une formule de Presburger. On voit que cette formule se résume à un système d'équations linéaire. Il serait bien de savoir construire un automate qui reconnaît notre formule. Pour cela il suffit de savoir construire un automate pour une équation linéaire pour pouvoir l'étendre aprés à un système d'équations.

Considérons l'équation suivante $x - y = 2$. On veut construire un automate qui reconnaît les solutions de cette équation. Pour cela, il faut d'abord préciser

que l'automate reconnaît les solutions encodées dans une certaine base qu'on va préciser. Le paragraphe qui suit explique comment passer du codage décimale au codage binaire, car on en aura besoin pour la suite.

Soit les vecteurs (3,1) et (7,5) solutions de notre équation. On va effectuer un codage binaire de ces solutions. Tout entier peut être encoder en base 2. Par exemple la décomposition binaire de 7 est 111 écrite de gauche à droite ; $7 = 2^2 + 2^1 + 2^0$. On peut'ajouter autant de 0 que possible dans l'écriture binaire de 7, c'est-à-dire 0000111 est un encodage binaire de 7. De même un vecteur d'entiers peut être encodé en base 2 en encodant chacune de ces composantes. Par exemple (7,5) est encodé par(7,5) =(111,101). Par la suite on alignera les codages de chaque composante sur le plus long en ajoutant les 0 nécessaires. Par exemple (3,1) = (11,1)=(11,01). Ces vecteurs binaires peuvent être vus comme des vecteurs de mots sur l'alphabet $\{0,1\}$. Par exemple le vecteur (111,101) se décompose en (1,1)(1,0)(1,1) sur l'alphabet des vecteurs de bit de dimension 2

$$\{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$$

en prenant les unités les plus à gauche pour chaque composante et les regrouper et ceci partant de la gauche vers la droite. L'automate associé à une équation linéaire accepte des mots construits sur $\{0,1\}^2$ en particulier et sur $\{0,1\}^n$ en général. Ces mots encodent une solution de cette équation. L'automate est construit comme suit pour l'équation $x - y = 2$:

1. Chaque état de l'automate à l'exception de l'état initial (égal à 0) correspond à un entier qui représente la valeur courante de la partie gauche de l'équation. Il suffit de donner les valeurs des vecteurs de bit à x et à y pour avoir la valeur courante.
2. De l'état correspondant à la valeur courante ci-dessus, que l'on note $\gamma = x - y$, il y a une seule transition pour chaque vecteur de bit conduisant à l'état $\gamma' = 2\gamma + \alpha$ où α est la nouvelle valeur courante.
3. Le seul état acceptant est celui qui conduit à la valeur 2, c'est-à-dire la partie droite de notre équation.

Formellement l'automate correspondant aux n-variables de l'équation $a.x = c$ est $A = (S, 2^n, \delta, s_i, c)$ où

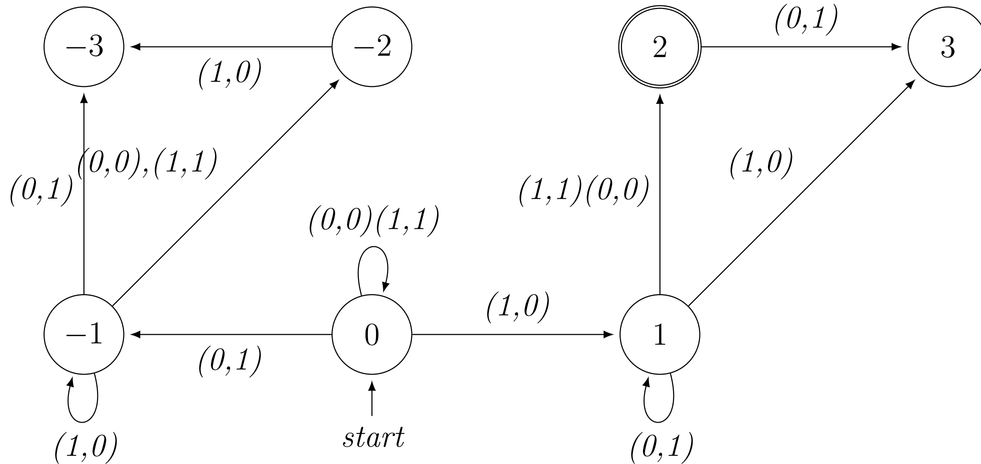
1. $S = Z \cup \{s_i\}$, c'est-à-dire les états sont des entiers plus un état spécial s_i ;
2. L'alphabet 2^n est l'ensemble des vecteurs de n-bits.
3. La fonction de transition δ définie par :
   (a) $\delta(s_i, b) = -a.b$ et
   (b) $\delta(\gamma, b) = 2\gamma + a.b$ pour $\gamma \neq s_i$
4. l'état initial s_i

5. le seul état final c.

Remarque. — On a remplacé x par $x' = 2x + b$ parce qu'on concaténe les bits de b à ceux de x et par suite on obtient le vecteur x' de longueur $k+1$ dont la valeur vérifie $x' = 2x + b$: le facteur 2 apparait du fait qu'on a décalé d'une unité vers la droite un nombre écrit en base 2 et on remplace les bits les plus à droite par ceux de b.

Tel qu'il est défini, l'automate est infini, mais il n'a qu'un nombre fini d'états co-acessibles, c'est-à-dire d'états à partir desquels on peut atteindre l'état final. En effet, si $||a||_1$ représente la norme du vecteur $a = (a_1, \ldots, a_n)$ définie par $||a||_1 = \sum_{i=1}^{n} |a_i|$, on a pour tout état γ tel que $|\gamma| > ||a||_1$, alors toute transition conduit à un état γ' tel que $|\gamma'| > |\gamma|$. Donc en particulier, tout état γ vérifiant $|\gamma| > ||a||_1$ et $|\gamma| > |c|$ n'est pas co-accessible.

Exemple 4.16. *Automate pour l'équation $x - y = 2$*



L'automate ainsi construit est déterministe et complet. Et on a la théorème suivant :

Théorème 4.17. *[WB00] L'automate ainsi construit est minimal. Il reconnaît les solutions de l'équation ax=c et peut être construit en temps et en espace exponentiel.*

Il est minimal dans le sens où les ensembles de mots acceptés par deux états différents ne peuvent pas être les mêmes. Pour plus de détails voir annexe 1.

Remarque. — Pour les inéquations de la forme $a.x \leq c$ on a la même construction. Les états acceptés sont tous ceux qui sont inférieurs à c et la terminaison est garantie.(voir annexe 1).

Nous avons montré précédemment comment construire un automate reconnaissant les solutions d'une équation linéaire. On construit par induction un automate reconnaissant toute formule de Presburger :

- $A_{\phi_1 \wedge \phi_2} = A_{\phi_1} \cap A_{\phi_2}$, l'automate reconnaissant le langage intersection construit en temps quadratique.
- $A_{\phi_1 \vee \phi_2} = A_{\phi_1} \cup A_{\phi_2}$, l'automate reconnaissant le langage union construit en temps linéaire. Cette opération fait perdre le caractère déterministe de l'automate.
- $A_{\neg\phi}$: l'automate reconnaissant le langage complémentaire de celui accepté par A_ϕ. Il peut être nécessaire de déterminiser l'automate A_ϕ pour cette opération ce qui peut engendrer un cout exponentiel.
- $A_{\exists x \phi}$ est l'automate correspondant à la projection de la coordonnée correspondant à x, opération faite en temps linéaire, mais qui peut faire perdre son caractère déterministe à l'automate.

Bibliographie

[BBB+07] E. Badouel, M.A. Bednarczyk, A.M. Borzyszkowski, B. Caillaud, and Ph. Darondeau. Concurrent secrets. *Discrete Event Dynamic Systems*, 17(4) :425–446, 2007.

[BD12] E. Badouel and M.L. Diouf. Opacité dans les systèmes workflows. In *CARI*, pages 70–77, 2012.

[BD14] E. Badouel and M.L. Diouf. Opacité des artefacts d'un système workflow. *Revue ARIMA*, 17 :177–196, 2014.

[BDA+04a] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, , and M. A. Musicante. Schema evolution for xml : A consistency-preserving approach. In vol. 3153 of Springer Lecture Notes in Computer Science, editor, *Mathematical Foundations of Computer Science*, pages 876–888, 2004.

[BDA+04b] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, and M. A. Musicante. Conservative extension of regular languages. In *International Conference of the Chilean Computer Science Society, number PR2200 in IEEE Proceedings*, pages 99–109, 2004.

[BKMR08] J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalised to transition systems. *Int. J. Inf. Sec.*, 7(6) :421–435, 2008.

[BKMW01] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree regular hedge languages over unranked alphabets, 2001. HKUST-TCSC-2001-05.

[BT08] E. Badouel and M. T. Tchendji. Merging hierarchically-structured documents in workflow systems. *Electronic Notes in Theoretical Computer Science*, 203(5) :3–24, 2008.

[BTNJF13] E. Badouel, R. Tchougong, C. Nkuimi-Jugnia, and B. Fotsing. Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition. *Theoretical Computer Science*, 480 :1–25, 2013.

[CC01] F. Chevalier and J. Chalopin. Représentation et algorithmique des ensembles semi-linéaires, 2001.

[CDG+08] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2008.

[CH09] D. Cohn and R. Hull. Business artifacts : A data-centric approach to modeling business operations and processes. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009.

[Cha] Manuel Chakravarty. The haskell 98 foreign function interface 1.0 ; an addendum to the haskell 98 report. haskell.org.

[CM01] J. Clark and M. Murata. Relax ng specification. http ://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001.

[CZ00] P. Caron and D. Ziadi. Characterization of glushkov automata. *TCS : Theoretical Computer Science*, 233 :75–90, 2000.

[DJM09] J. Dubreil, T. Jéron, and H. Marchand. Monitoring confidentiality by diagnosis techniques. In *European Control Conference*, pages 2584–2590, Budapest, Hungary, August 2009.

[Dua05] D. Duarte. *Une méthode pour l'évolution de schémas XML préservant la validité des documents*. PhD thesis, Université François Rabelais-Tours, 2005.

[Gin66] Seymour Ginsburgh. *The Mathematical Theory of Context-Free Languages*. Mc Graw-Hill, 1966.

[GS66a] S. Ginsburgh and E.H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2) :285–296, 1966.

[GS66b] S. Ginsburgh and E.H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2) :285–296, 1966.

[HS04] D. Hughes and V. Shmatikov. Information hiding, anonymity and privacy : a modular approach. *Journal of Computer Security*, 12(1) :3–36, 2004.

[HVP05] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1) :46–90, 2005.

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecturea. *Trans. Am. Math. Soc. (1960)*, 95 :210–225, 1960.

[Lin11] F. Lin. Opacity of discrete event systems and its applications. *Automatica*, 47(3) :496–500, 2011.

[ON01] O.Danvy and L. R. Nielsen. Defunctionalization at work. Technical report, BRICS RS-01-23, 2001.

[OSG08] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. December 2008.

[Pug92] W. Pugh. The omega test : A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, pages 102–114, 1992.

[Reu66] Christophe Reutenauer. *Aspects mathématiques des réseaux de Petri.* Mc Graw-Hill, 1966.

[Vea97] Margus Veanes. On computational complexity of basic decision problems of finite tree automata. Technical Report 133, Uppsala University, Computing Science Department, January 1997.

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In vol. 1785 of Springer Lecture Notes in Computer Science, editor, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–19, 2000.

Résumé

Une propriété d'un objet est dite *opaque* pour un observateur si celui-ci ne peut déduire que la propriété est satisfaite sur la base de l'observation qu'il a de cet objet. Supposons qu'un certain de nombre de propriétés (appelées *secrets*) soient attachées à chaque intervenant d'un système, nous dirons alors que le système lui-même est opaque si chaque secret d'un observateur lui est opaque : il ne peut percer aucun des secrets qui lui ont été attachés. L'opacité a été étudiée préalablement dans le contexte des systèmes à événements discrets où différents jeux d'hypothèses ont pu être identifiés pour lesquels on pouvait d'une part décider de l'opacité d'un système et d'autre part développer des techniques pour diagnostiquer et/ou forcer l'opacité. Cette thèse constitue la première contribution au problème de l'opacité des artefacts d'un système à flots de tâches (système workflow). Notre propos est par conséquent de formaliser ce problème en dégageant les hypothèses qui doivent être posées sur ces systèmes pour que l'opacité soit décidable. Nous indiquons quelques techniques pour assurer l'opacité d'un système.

Abstract

A property (of an object) is opaque to an observer when he or she cannot deduce the property from its set of observations. If each observer is attached to a given set of properties (the so-called secrets), then the system is said to be opaque if each secret is opaque to the corresponding observer. Opacity has been studied in the context of discrete event dynamic systems where technique of control theory were designed to enforce opacity. This thesis is the first attempt to formalize opacity of artifacts in data-centric workflow systems. We motivate this problem and give some assumptions that guarantee the decidability of opacity. Some techniques for enforcing opacity are indicated.