Lot 4.2

Technologie de modélisation

Probabilités

Stratégies de réécriture probabiliste dans ELAN4

Description :	Ce rapport décrit l'implantation des stratégies de réécriture probabilistes dans le prototype ELAN 4, il rappelle l'apport de ces stratégies et présente quelques exemples.
Auteur(s) :	Olivier BOURNEZ, Florent GARNIER, Claude KIRCHNER
Référence :	AVERROES / Lot 4.2 / Fourniture 5 / V1.1
Date :	septembre 2004
Statut :	validé
Version :	1.1

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (Ecole Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

1 Introduction

Il manquait jusqu'à présent un moyen de modéliser ou d'exprimer des choix ou des comportements aléatoires au sein de la réécriture avec stratégie. C'est pour exprimer ce type d'informations que nous avons introduit le concept intuitif de stratégies de réécriture probabiliste. Ces stratégies ont pour but de permettre de décrire au niveau transitoire une distribution de probabilité sur un ensemble de choix possibles.

À cause d'un ensemble de phénomène déjà présents au niveau de la réécriture avec stratégie "classique", déterminer quel sera le comportement général d'un système de réécriture avec stratégies probabilistes est très difficile, une approche est cependant proposée dans [5]. En effet, la réécriture avec stratégies, et la réécriture conditionnelle, introduisent des phénomènes non déterministes, empêchant dans le cas général d'obtenir des résultats de nature probabilistes tels que ceux existants pour les processus stochastiques. Afin de pouvoir observer l'évolution et le comportement de tels systèmes de réécriture, nous avons ajouté ces stratégies au sein d'un logiciel déjà existant, ELAN 4. ELAN 4 est une version complètement réécrite du logiciel ELAN 3. Ce logiciel est le fruit de la coopération de deux équipes, SEN1 au CWI d'Amsterdam et de l'équipe PROTHEO du LORIA à Nancy. Ce logiciel à été développé dans l'optique de pouvoir prototyper des composants nouveaux. C'est pour cette raison qu'ELAN 4 à été choisi pour expérimenter les stratégies de réécriture probabilistes.

Pour bien cerner le contexte dans lequel se situe ce travail, nous allons dans un premier temps présenter les logiques de réécriture ainsi que les fondements de la réécriture classique. Une fois cette présentation faîte, il sera possible de présenter sans ambiguité notre définition des stratégies de réécriture ainsi que leur sémantique.

2 Brève présentation des logiques de réécriture.

Dans un système de réécriture, les stratégies de réécriture permettent d'exprimer un sous ensemble des réductions possibles d'un terme. Certains de ces sous ensembles peuvent être décrits grâce à un langage d'expression de stratégies. On peut par exemple spécifier des ordres de parcours, tenir compte d'éventuels échecs d'application de stratégie ou de réduction de termes.

Pour présenter correctement ce que sont les stratégies de réécriture, puis les stratégies de réécriture probabilistes, nous devons tout d'abord présenter ce qu'est la logique de réécriture introduite dans [11], puis définir les systèmes de calcul comme des théories de réécriture dans la logique de réécriture où les stratégies servent à limiter l'espace de calcul.

2.1 Quelques définitions et notations

Nous reprenons ici les notions ainsi que les notations utilisées dans [9] et [3].

Notation 1 $\mathcal{T}(\Sigma, X)$

Soit Σ une signature, soit X un ensemble de variables tel que $\Sigma \cap X = \emptyset$. La notation $\mathcal{T}(\Sigma, X)$ représente l'ensemble des Σ-termes sur X.

Dans les logiques de réécriture, les preuves sont des objets du premier ordre et sont représentées par des "termes de preuve". Soit $\mathcal{F}$ un ensemble de fonctions, on note par $\mathcal{F}_n$ l'ensemble des fonctions arité n de $\mathcal{F}$, en d'autres termes $\mathcal{F} = \cup_{n \in \mathbb{N}} \mathcal{F}_n$. Soit X un ensemble de variables, $\mathcal{T}(\mathcal{F}, X)$ est l'algèbre des termes du premier ordre construits à partir de la signature $\mathcal{F}$ et de l'ensemble des variables X.

Nous allons voir par la suite que l'une des entités élémentaire des langages d'expression de stratégie consiste à nommer un certain ensemble de régles de réécriture pour pouvoir les appeler explicitement. Pour pouvoir effectuer ce nommage, on va se munir de $\mathcal{L}$ un ensemble de symboles (étiquettes) .

Définition 1 ***Terme de preuve***
Un terme de preuve est un terme construit sur les l'ensemble des termes $\mathcal{T}(\mathcal{F}, X)$, les symboles de fonctions $\mathcal{F}$, les symboles $\mathcal{L}$ ainsi que l'opérateur de concaténation ";". Plus formellement un terme de preuve est un élément de l'algèbre $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{;\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, X))$

Les termes de preuves servent à représenter des ensembles de calculs, ou de preuves. On note par $t \Rightarrow t'$ le fait que t se réécrive en une seule étape en t'. Une étape de réécriture sert à représenter une étape de calcul élémentaire. Une preuve par exemple, sera représentée par un terme de preuve qui sera lui même la concaténation d'un ensemble de symboles représentant des étapes élémentaires de calculs, appelées séquents.

Ainsi on notera par la suite $t \overset{\pi}{\Rightarrow} t'$ le fait que le terme t se réécrive en t' en suivant une succession d'étapes de réécriture notée π.

2.2 Logique de réécriture

Une logique est définie par une syntaxe, par un système de déduction, une classe de modèles, et une relation de satisfaction. Les quatre composants respectivement notés $(Synth, \vdash, Model, \models)$ vont être décrits ci dessous.

Syntaxe

La syntaxe nécessaire à la définition d'une logique est donnée par une signature définissant l'ensemble des construction syntaxiques valides. Afin de rester général, considérons $Synt$ une classe de paires (Σ, sen) associant Σ à une application reliant elle même la signature Σ à l'ensemble des constructions syntaxiques valides faite à partir de cette signature.

Dans les logiques de réécritures, une signature consiste en un triplet $\Sigma = (\mathcal{L}, \mathcal{F}, E)$ où $\mathcal{F}$ et E sont des ensembles de symboles de fonctions et E est un ensemble de $\mathcal{T}(\mathcal{F}, \mathcal{X})$-égalités. Les égalités dans E concernent les axiomes structuraux permettant de construire la signature Σ. Par exemple, dans une signature contenant l'opérateur de conjonction $\wedge$, E contient les axiomes d'associativité et de commutativité concernant $\wedge$. Les objets construits à partir de cette signature définissent les séquentis de la forme $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ où $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\pi \in \mathcal{PT}$

Intuitivement, on retiendra que de tels mots signifient que la preuve π permet de déduire t' à partir de t.

Systèmes de déduction

Afin de construire le système de déduction de la logique de réécriture, on introduit d'abord la notion de théorie de réécriture. Pour certaines classes de syntaxe $Synt$ et $(\Sigma, sen) \in Synt$, une théorie est un couple $T = (\Sigma, \Phi)$ où $\Phi \subseteq sen(\Sigma)$. On dit dans ce cas que T est représentée par l'ensemble d'axiomes Φ.

Pour une signature Σ, un système de déduction est une représentation abstraite de la relation de satisfaisabilité de l'assertion ϕ connaissant un ensemble d'axiomes Φ et en utilisant le règles de la logique. Formellement, étant donnée une classe de syntaxe $Synt$, un système de déduction est une paire $(Synt, \vdash)$ telle que $\vdash$ soit une fonction associant à chaque $(\Sigma, sen) \in Synt$ une relation binaire $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ vérifiant les aspects présentés dans la Figure 1

$\forall \phi \in sen(\Sigma), \phi \vdash_\Sigma \phi$	(Réflexivité)
Si $\Phi \vdash_\Sigma \phi$ et $\Phi \subseteq \Phi'$ alors $\Phi' \vdash_\Sigma \phi$,	(Monotonie)
Si $(\forall i \in I, \Phi \vdash_\Sigma \phi_i)$ et $(\Phi \cup_{i \in I} \phi_i) \vdash_\Sigma \phi$,	(Transitivité)
Pour tout morphisme de signatures H : si $\Phi \vdash_\Sigma \phi$ alors $H(\Phi) \vdash_{H(\Sigma)} H(\Phi)$	(Translation)

FIG. 1 – Régles des systèmes de déduction

Dans les logiques de réécritures, la notion de théorie de réécriture a été introduite pour exprimer et construire les systèmes de déduction. De même un système de déduction approprié permet de définir par induction la relation de déduction.

Une théorie de réécriture étiqueté, est un 5 uplet $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, \mathcal{R})$ où $\mathcal{X}$ est un ensemble infini et dénombrable de variables, $\mathcal{L}$ et $\mathcal{F}$ sont deux ensembles de symboles de fonctions, E un ensemble de $\mathcal{T}(\Sigma, \mathcal{X})$-égalités, et R un ensemble de règles de réécriture nommées, de la forme $\ell : g \rightarrow d$ où $\ell \in \mathcal{L}$ et $g, d \in \mathcal{T}(\Sigma, \mathcal{X})$ et $\mathcal{V}ar(d) \subseteq \mathcal{V}ar(g)$, et où l'arité de ℓ est le nombre de variables distinctes de g. L'usage des étiquettes pour nommer les règles est crucial pour définir les stratégies et les modèles de la logique de réécriture où le nommage des régles permet de suivre les étapes de déduction.

Les définitions peuvent être étendues au cadre de la réécriture conditionnelle au prix d'une complication du formalisme utilisé. Le Lecteur peut consulter [11].

Une théorie de réécriture étiquetée $\mathcal{R}$ implique le séquent $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ (On note ce fait $\mathcal{R} \vdash \pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$) si $\pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$ est obtenu grâce à

Réflexivité

$\twoheadrightarrow$
$\langle t\rangle_E : \langle t\rangle_E \to \langle t\rangle_E$
si $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$

Congruence

$\pi_1 : \langle t_1\rangle_E \to \langle t'_1\rangle_E, \ldots, \pi_n : \langle t_n\rangle_E \to \langle t'_n\rangle_E$
$\twoheadrightarrow$
$f(\pi_1,\ldots,\pi_n) : \langle f(t_1,\ldots,t_n)\rangle_E \to \langle f(t'_1,\ldots,t'_n)\rangle_E$
Si $f \in \mathcal{F}_n$

Remplacement

$\pi_1 : \langle t_1\rangle_E \to \langle t'_1\rangle_E, \ldots, \pi_n : \langle t_n\rangle_E \to \langle t'_n\rangle_E$
$\twoheadrightarrow$
$l(\pi_1,\ldots,\pi_n) : \langle g(t_1,\ldots,t_n)\rangle_E \to \langle d(t'_1,\ldots,t'_n)\rangle_E$
Si $l : g(x_1,\ldots,x_n) \to d(x_1,\ldots,x_n) \in R)$

Transitivité

$\pi_1 : \langle t_1\rangle_E \to \langle t_2\rangle_E \to \langle t_3\rangle_E$
$\twoheadrightarrow$
$\pi_1;\pi_2 : \langle t_1\rangle_E \to \langle t_3\rangle_E$

FIG. 2 – Régles de déduction

une combinaison finie des opérations décrites dans la Figure 2.

Remarque : A cause des régles de réflexivité et de remplacement, si la règle $r = (l : g \to d) \in R$, alors le séquent $seq(r) = (l(\langle x_1\rangle_E,\ldots,\langle x_n\rangle_E) : \langle g\rangle_E \to \langle d\rangle_E)$ peut être dérivé. L'ensemble des séquents $\{seq(r)|r \in R\}$ est noté $seq(R)$.

Le système de déduction est ainsi défini pour une signature $\Sigma = (\mathcal{L},\mathcal{F},\mathcal{X},E)$ et une théorie de réécriture étiquetée $\mathcal{R}$, par $seq(R) \vdash_\Sigma (\pi : u \to v) si \mathcal{R} \vdash (\pi : u \to v)$.

Modèles Pour une syntaxe donnée $Synt$, pour chaque signature et chaque ensemble de constructions syntaxiques valides $(\Sigma, sen) \in Synt$ est associée une classe $Mod(\Sigma)$ d'objets appelés $\Sigma-$Modèles et une relation de satisfaction notée $\models_\Sigma \subseteq Mod(\Sigma) \times sen(\Sigma)$ compatible avec les morphismes de signatures.

Pour un ensemble de constructions syntaxiques Φ, $Mod(\Sigma,\Phi)$ est la classe de tous les $\Sigma-$modèles M tels que $\forall\phi \in \Phi, M \models_\Sigma \phi$, appelés modèles satisfaisant Φ. $Mod(\Sigma,\Phi) \models_\Sigma \phi$ signifie que $M \models_\Sigma \phi$ quel que soit $M \in Mod(\Sigma,\Phi)$

Un modèle de logique de réécriture peut être choisi comme l'espace des calculs de la théorie de réécriture $\mathcal{R}$. Un tel espace est déterminé par l'ensemble des termes de preuves π des séquents $\pi : \langle t\rangle_E \mapsto \langle t'\rangle_E$ modulo les relations d'équivalences de calcul. Ces équivalences sont décrites dans l'ensemble E et dans l'ensemble $\mathcal{E}_{\mathcal{PT}(R)}$ des axiomes équationnels de l'ensemble des termes de preuves décrits dans la figure 3. Ainsi le modèle pris en compte est l'ensemble quotient suivant :

$$\{\pi|\mathcal{R} \vdash \pi : \langle t\rangle_E \to \langle t'\rangle_E\}/(E \cup \mathcal{E}_{\mathcal{PT}(R)})$$

.

$\forall \pi_1, \pi_2, \pi_3 \in \mathcal{PT}\ \pi_1; (\pi_2; \pi_3) = (\pi_1; \pi_2); \pi_3$	Associativité
$\forall \pi : \langle t \rangle_E \rightarrow \langle t' \rangle_E$, et $\langle t \rangle_E; \pi = \pi \langle t \rangle_E; \pi = \pi$	Identité locale
$\forall f \in \mathcal{F}_n, n \in \mathbb{N}, \forall \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_n :$ $f(\pi_1; \pi'_1, \ldots, \pi_n; \pi'_n) = f(\pi_1, \ldots, \pi_n); f(\pi'_1, \ldots, \pi'_n)$	Indépendance
$\forall f \in \mathcal{F}_n, n \in \mathbb{N} :$ $f(\langle t_1 \rangle_E, \ldots, \langle t_n \rangle_E) = \langle f(t_1, \ldots, t_n) \rangle_E$	Préservation de E
$\forall l : g \rightarrow d \in R, \forall \pi_1 : \langle t_1 \rangle_E \rightarrow \langle t'_1 \rangle_E, \ldots, \pi_n : \langle t_n \rangle_E \rightarrow \langle t'_n \rangle_E$ $l(\pi_1, \ldots, \pi_n) = l(\langle t_1 \rangle_E, \ldots, \langle t_n \rangle_E); d(\pi_1, \ldots, \pi_n)$ et $l(\pi_1, \ldots, \pi_n) = g(\pi_1, \ldots, \pi_n); l(\langle t'_1 \rangle_E, \ldots, \langle t'_n \rangle_E)$	Parallel Move Lemma

FIG. 3 – $\mathcal{E}_{\mathcal{PT}(R)}$:Equivalence des termes de preuve

Logiques

Nous avons maintenant défini les éléments d'une logique $\mathcal{L}$, qui est donnée par cinq composants $\mathcal{L} = (Synt, sen, \mathcal{M}od, \vdash, \models)$ tels que $(Synt, sen, \vdash)$ est un système de déduction, et $(Synt, sen, \mathcal{M}od, \models)$ une classe de modèles satisfaisant la condition de correction :

$$\forall (\Sigma, sen) \in Synt, \Phi \subseteq sen(\Sigma), \phi \in sen(\Sigma), \Phi \vdash_\Sigma \phi \Rightarrow Mod(\Sigma, \Phi) \models_\Sigma \phi$$

La logique est dite complète lorsque la précédente implication est une équivalence.

Calcul de preuve

Nous venons de définir de manière formelle ce qu'est une logique, sans apporter d'information sur la structure du système de déduction. Il nous faut maintenant décrire la manière avec laquelle prouver des propriétés à partir d'un ensemble d'axiomes Φ. Une telle description des preuves doit être suffisamment flexible pour permettre de définir un "sous" calcul permettant de prouver des assertions dans un sous ensemble de propriétés.

Il est nécessaire qu'un calcul de preuve décrive la relation $\vdash$.

On remarque qu'il peut exister plusieurs calculs de preuve pour une logique donnée, comme c'est le cas dans la logique du premier ordre.

Un calcul de preuve associe à chaque théorie T une structure algébrique $P(T)$ utilisant les axiomes de T comme hypothèses ainsi que les régles de déduction comme opérateurs pour construire de nouvelles preuves.

On note par $proofs(T)$ l'ensemble de toutes les preuves de tous les théorèmes de la théorie T. On associe à chaque preuve le théorème qu'elle prouve, τ_T associe chaque preuve $p \in proofs(T)$ la propriété ϕ dont p est une preuve, $\phi = \tau_T(p)$. Un sous calcul de preuve formalise une restriction d'un système de déduction à une syntaxe spécifique et restreint de même l'ensemble des axiomes ainsi que des des conclusions.

Un calcul de preuve ou un sous calcul est formellement représenté par un système de déduction, et pour chaque théorie T, le triplet associé $(P(T), proofs(T), \tau_T)$.

Dans les logiques de réécriture, l'ensemble des preuves est définie comme l'ensemble des termes de preuves, et le calcul de preuves suit les mêmes régles que celles du système de déduction, définies en Figure 1.

Calculs

Le principal ingrédient d'un système de calcul, est une théorie de réécriture étiqueté, à partir de laquelle on définit la notion de calcul.

Définissons dans un premier temps ce qu'est une simple étape de calcul élémentaire. Pour une théorie de réécriture $\mathcal{R}$, pour le séquent $\pi : \langle t \rangle_E \to \langle t' \rangle_E$ avec le terme de preuve $\pi = t[l(\sigma\overline{x})]_\omega$

Alors

$$\langle t \rangle_E \Rightarrow_{l,\sigma,\omega} \langle t' \rangle_E$$

est appelé un pas simple de réécriture. On remarque que cela correspond à la définition habituelle de l'application de la règle de réécriture nommée l à l'occurrence ω en utilisant la filtration σ.

Plus précisément, nous allons nous intéresser aux calculs équivalents modulos les axiomes $\mathcal{E}_{\mathcal{PT}}(R)$, le résultat présenté dans [11] montre que l'on peut en effet procéder de cette manière, grâce à l'associativité de l'opérateur de concaténation " ;", n'importe quel séquent peut être décomposé en une composition de séquents élémentaires.

En d'autres termes, $\forall \pi$ séquent, soit $\pi = \langle t \rangle_E = \langle t' \rangle_E$, plus généralement $\exists n \in \mathbb{N}$

$$\langle t \rangle_E \Rightarrow_{\ell_0} \langle t_1 \rangle_E \Rightarrow_{\ell_1} \ldots \Rightarrow_{\ell_{n-1}} \langle t' \rangle_E$$

avec $\pi =_{A(;)} (\pi_0; \pi_1; \ldots; \pi_{n-1})$, où $A(;)$ dénote l'associativité de " ;". Lorsque π est le précèdent terme de preuve, $\langle t_n \rangle_E$ est le résultat de l'application de π sur $\langle t_0 \rangle_E$, et s'écrit $\langle t \rangle_E \stackrel{\pi}{\Rightarrow} \langle t' \rangle_E$

La relation d'équivalence générée par $(E \cup \mathcal{E}_{\mathcal{PT}(R)})$ sur les termes de preuve induit une relation d'équivalence sur les calculs, deux calculs sont équivalents (ils donnent le même résultat), si leur terme de preuve sont équivalents.

Stratégies de réécriture Les stratégies contrôlent l'application des régles de réécriture en spécifiant des parcours dans l'arbre de toutes les dérivations possibles, et de cette façon décrivent quels sont les noeuds considérés comme des résultats d'un calcul.

Définition 2 ***Stratégie de réécriture***

Étant donné qu'une stratégie est un sous ensemble de l'ensemble des termes de preuves construits par concaténation, si S est une stratégie, t est un Σ-terme, on note par $S(t)$ l'application de la stratégie S contre le terme t, c'est-à-dire l'ensemble des dérivations de S à partir de t.

$$S(t) = \{t' | \exists \pi \in S, t \stackrel{\pi}{\Rightarrow} t'\}$$

Nous allons également introduire la notion d'échec. Si $S(t) = \emptyset$, alors on parle de l'échec de l'application de la stratégie S contre le terme t. Cela signifie simplement qu'aucun terme de preuve de S permet de dériver t vers un autre t'.

Langages de description de stratégies

Une manière de définir des stratégies, est d'énumérer l'ensemble des sous parties des termes de preuves. En pratique, cela n'est en général pas possible, car l'ensemble des sous parties des termes de preuve n'est pas forcément fini ni dénombrable.

Pour pouvoir travailler avec ces stratégies, un langage de description va être nécessaire afin de choisir les sous ensembles de termes de preuves avec lesquels nous allons travailler. Le langage de description de stratégies va être un moyen syntaxique d'exprimer la sémantique de ces stratégies de réécriture sous la forme d'un programme.

2.3 Brève présentation du langage de description de stratégies utilisé en ELAN

Nous présentons ici, l'implantation des stratégies de réécritures probabilistes dans le prototype de la version 4 du système ELAN [2]. Les stratégies de réécriture probabilistes s'ajoutent aux stratégies de réécriture [3] afin de décrire certains phénomènes probabilistes. Cette notion de stratégie de réécriture probabiliste à été présentée dans [4] et [5].

2.3.1 Le langage de description de stratégies non probabilistes d'ELAN

Nous allons présenter brièvement les différents opérateurs de base composant le langage de description de stratégies d'ELAN.

- `dk` ou don't know. La stratégie `dk`$(S_1, \ldots, S_n)$ donne tous les résultats de l'application de toutes les stratégies $S_1, \ldots, S_n$. Si tous les S_k échouent, alors `dk` échoue.
- `dc` ou don't care. La stratégie `dc`$(S_1, \ldots, S_n)$ donne tous les résultats de l'application **d'une stratégie** parmi $S_1, \ldots, S_n$ n'échouant pas si elle existe. Si tous les S_k échouent, alors `dc` échoue. Le choix est non déterministe.
- `first`. La stratégie `first`$(S_1, \ldots, S_n)$ donne tous les résultats de l'application de la première stratégie n'échouant pas. Si toutes les stratégie échouent, alors `first` échoue.
- `fail`. La stratégie échouent toujours.
- La stratégie `;`. S_1 `;`S_2 correspond à l'application de S_2 conditionnellement au succès de S_1 et correspond à l'axiome de transitivité de la logique de réécriture.
- `Id` est la stratégie identité, elle retourne le même terme d'entrée, et peut toujours être appliquée.

Ces stratégies, permettent de sélectionner, un sous ensemble de stratégies. Les stratégies en ELAN 4 font appel à des règles de réécriture nommées. Ces

règles peuvent s'appliquer si et seulement si elles sont explicitement appelées, via une stratégie.
Par exemple, prenons un ensemble de règles très simple, comme celui présenté dans la Figure 2.3.1 et appliquons `(dc(r1,r2))` sur le terme `A` nous obtiendrons soit `B` ou `C`, mais pas `D` car la règle `[r3]` ne sera pas appelée.

Ces stratégies sont formellement décrites dans [1] et [6].

```
[r1] A => B
[r2] A => C
[r3] A => D
```

FIG. 4 – Unsystème de réécriture simple

2.4 Les stratégies probabilistes

2.4.1 Concept

Les stratégies de réécriture probabiliste, peuvent être vues comme la stratégie `dc` à laquelle ont rajoute une information de nature probabiliste quand à la manière de choisir les stratégies passées en paramètre. Le choix pour la stratégie `dc` non déterministe, alors que le choix pour les stratégies de réécriture probabilistes devront suivre une distribution de probabilité. Le processus de choix probabiliste est donc une restriction du processus de choix non déterministe.

Les stratégies de réécriture probabiliste vont nous servir dans un premier temps à modéliser le comportement de certains automates qui eux même vont nous permettre de simuler l'exécution de certains systèmes, comme par exemple, un ensemble de machines dialoguant en utilisant le protocole CSMA/CD. Ce que permettent de faire les stratégies de réécriture probabilistes, n'est autre que de mettre une distribution probabiliste sur un ensemble de stratégies de réécritures (probabilistes ou non).

Nous avons implanté dans le prototype deux stratégies de réécriture probabiliste, `pc` et `pa`, pour probabilistic choice et probabilistic accept. L'idée de la première stratégie est d'associer à chaque stratégie passée en paramètre un poids. `pa` exprime un choix entre l'application d'une stratégie et la stratégie codant l'échec `fail`.

2.4.2 Sémantique de `pc` et `pa`

Ces deux opérateurs de stratégies décrivent une manière de choisir les stratégies passées en paramètre.

Définition 3 ***L'opérateur*** `pc`
`pc` est une stratégie de réécriture où l'on associe à l'ensemble des stratégies passées en paramètre une distribution de probabilité. `pc` prend pour argument une liste de couples (S, p) où S est une stratégie, et p est un entier.

La probabilité que $\mathtt{pc}((S_1, p_1), \ldots, (S_n, p_n))$ *s'évalue en* S_i *vaut*

$$\frac{p_i}{\sum_{k=1}^{n} p_k}$$

On définit de la même manière l'opérateur de stratégie `pa` :

Définition 4 ***L'opérateur*** `pa`
`pa` *prend deux arguments, une stratégie et un nombre rationnel positif inférieur à un.* $\mathtt{pa}(S, p : q)$ *évalue en* S *avec la probabilité* $\frac{p}{q}$ *et s'évalue en* $\mathtt{fail}$ *avec la probabilité* $1 - \frac{p}{q}$*. On a donc* $\mathtt{pa}(S, p : q) = \mathtt{pc}((S, p), (\mathtt{fail}, q - p))$.

2.4.3 La syntaxe des stratégies des opérateurs `pc` et `pa`

La syntaxe permettant de les utiliser est simple et directement inspirée des définitions de `pa` et `pc` données lors du paragraphe précédent. Par exemple, soient $S_1 \ldots S_n$ n stratégies et $p_1 \ldots p_n$ n entiers, alors :

```
 ( pc(S1:p1 ; S2:p2 ; ... ;Sn:pn ) ) t
```

signifie que l'on associe le poids pk à S_k.

Notons qu'en ELAN, l'opérateur d'application de stratégie est placé entre parenthèse à gauche du terme à réduire.

La syntaxe de l'opérateur `pc` est la suivante : `pc` prend en paramètre une liste de couples "stratégies, entiers", séparés par un double point, un point virgule sépare chaque couple. Comme il est expliqué dans le paragraphe précédent, la probabilité que la stratégie S_k s'applique sur le terme T vaut $\frac{p_k}{\sum_{i=1}^{n} p_i}$

On utilise une syntaxe semblable pour l'opérateur `pa` :

```
(pa(S:7:9)) t
```

`pa` prend en argument trois paramètres, une stratégie et deux entiers. Ici la stratégie S sera appliquée sur le terme t avec une probabilité $\frac{7}{9}$, `fail` sera appliquée avec probabilité $\frac{2}{9}$.

3 Implémentation des opérateurs de stratégies probabilistes dans ELAN 4

Les deux stratégies sont implémentées dans le prototype ELAN 4. L'implantation s'effectue en deux phases principales :
– déclaration de syntaxe,
– intégration des stratégies au sein de l'interpréteur.

3.1 Intégration de la syntaxe.

Cette partie de développement consistait à reconnaître à partir d'une définition de syntaxe, l'ensemble des stratégies ainsi que la distribution associée. Ceci en utilisant la technologie SDF [7] développée au CWI à Amsterdam de la manière suivante :

Un fichier SDF contient la définition de syntaxe des opérateurs :

```
module stratégies/ProbStrategies[StratSort]

imports
        basic/BuiltinInt
hiddens
  sorts StratSort
        ProbStratSort
exports
  context-free syntax

        StratSort ":" BuiltinInt -> ProbStratSort

    "pc" "(" {ProbStratSort ","}+ ")" ->
                        StratSort {stratop, builtin("pc"), cons("pc")}
    "pa" "(" StratSort ":" BuiltinInt ":" BuiltinInt  ")" ->
                StratSort {stratop, builtin("pa"), cons("pa")}
```

On définit ici deux opérateurs, `pa` et `pc`. L'opérateur `pc` prend en paramètre une liste de `ProbStratSort`. Le type probstratsort code le couple "stratégie, entier", on définit ici que la syntaxe d'un `ProbStratSort` est une stratégie séparée d'un entier par un point virgule.

Lors de sa compilation, ELAN 4 fait appel à l'outil ApiGen [8] [13], toujours développé au CWI et au LORIA, lui permettant de générer automatiquement un jeu de fonctions manipulant les types définis ci dessus. Ainsi, ApiGen nous permet d'avoir toutes les fonctions de manipulation de types écrites en C.

Le travail effectué par ApiGen simplifie drastiquement le processus d'intégration de la partie traitant l'interprétation des stratégies au sein du code de l'interpréteur ELAN 4. Il suffit de rajouter dans l'interpréteur une fonction traitant la stratégie `pc` (resp `pa`). Les paramètres sont récupérés sous la forme d'une liste et on en extrait les valeurs grâce aux fonctions générées par ApiGen. A partir de ces informations, les distributions passées en paramètres sont recalculées de manière classique à partir d'une distribution uniforme obtenue grâce à un générateur de nombre pseudo-aléatoire, en l'occurrence la fonction `random` de la librairie C.

3.2 Génération des distributions

La fonction `random`, calcule toujours la même suite de nombre pour une graine fixée, elle a été choisie pour des raisons de facilite de développement et sera remplacée par un autre générateur de nombre aléatoires dans les versions ultérieures d'ELAN 4. L'annexe décrit la génération de nombre aléatoire implanté. Le remplacement de `random` par un générateur de nombre pseudo aléatoire non reproductible tel que le système HAVEGE [12] permettra d'obtenir des suites de nombres différentes d'une exécution à l'autre, soit en calculant à chaque exécution une nouvelle graine pour un bon générateur de nombre pseudo aléatoires dont les caractéristiques sont bien connues, ou bien en utilisant directement les valeurs calculées par le système physique si on à confiance en ses qualités.

4 Exemples de codages d'automates probabilistes

Grâce aux stratégies de réécriture probabilistes, nous pouvons coder des automates probabilistes, familles particulièrement utiles pour la modélisation de systèmes complexes où interviennent des phénomènes aléatoires. Mais, donnons une série d'exemples et commençons par l'incontournable pile ou face. Il se code en trois lignes seulement, comme le montre la Figure 4

```
rules

[] pileface => pc( pile:1 ; pile:1 )
[pile] coin => tail
[face] coin => head
```

FIG. 5 – fichier pilouface.elan

Ici, le terme `pc( pile :1 ; pile :1 ) coin` va se réécrire de manière équiprobable en `head` ou `tail`

4.1 Un automate probabiliste simple

L'automate suivant peut être codé en ELAN 4, comme le montre l'exemple ci dessous.

```
rules
[RA] A => X where X := (RB) B
[RB] B => X where X := ( pc(BA :5, BC :1) ) B

[BA] B => X where X := (RA) A
[BC] B => C
```

4.2 Une marche aléatoire particulière

Il est également possible de coder des marches aléatoires un peu particulières, comme celle suivant les contraintes suivantes :

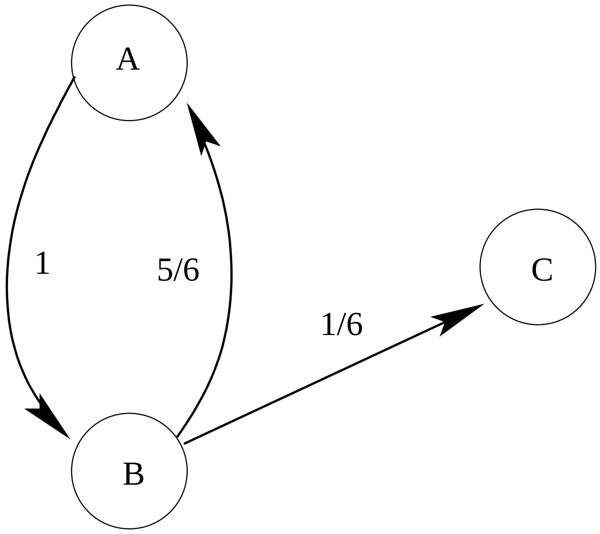

FIG. 6 – A probabilistic automaton

$(X_n, Y_n) \in \mathbb{N}^2$
$\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n + 1, Y_n)) = \mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n, Y_n + 1)) = \frac{1}{6}$ si $X_n \neq 0$ et $Y_n \neq 0$ $\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n - 1, Y_n)) = \mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n, Y_n - 1)) = \frac{1}{3}$ si $X_n \neq 0$ et $Y_n \neq 0$
$\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n - 1, Y_n)) = \frac{2}{3}$ si $X_n \neq 0$ et $Y_n = 0$ $\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n + 1, Y_n)) = \frac{1}{3}$ et $X_n \neq 0 et Y_n = 0$
$\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n, Y_n - 1)) = \frac{2}{3}$ et $X_n = 0 et Y_n \neq 0$ $\mathbb{P}((X_{n+1}, Y_{n+1}) = (X_n + 1, Y_n)) = \frac{1}{3}$ et $X_n = 0 et Y_n \neq 0$
$(0, 0)$ état terminal

Le graphique présenté dans la figure 7 permet d'illustrer les mouvements possibles en fonction de la position du point sur $\mathbb{N}^2$.

La position sur $\mathbb{N}^2$ est représentée par un point bleu. On distingue quatre types de positions, la position terminale $(0, 0)$ d'où l'on ne bouge plus, les bords que l'on ne peut pas quitter et enfin les autres points où l'on peu monter, descendre, aller à gauche ainsi qu'à droite.

La longueur des flèches est proportionelle à la probabilité d'effectuer le mouvement correspondant. Les flèches rouges correspondent aux mouvement éloignant le point de l'origine, les flèches vertes représentent les mouvements le rapprochant de l'origine.

On sait classiquement que presque sûrement on atteint l'état terminal $(0, 0)$, on peut dire que ce système termine presque sûrement. voici comment coder une telle marche en ELAN 4 :

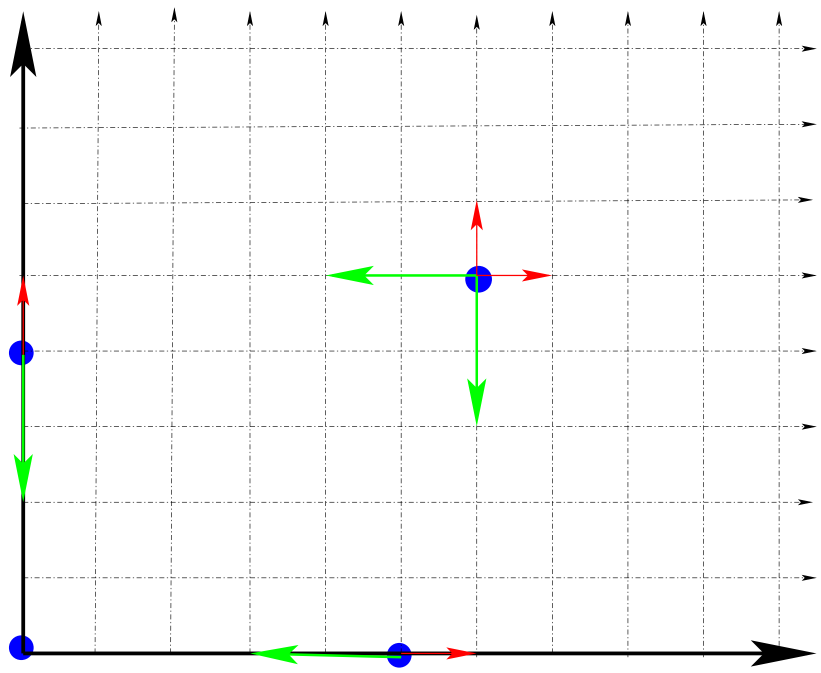

FIG. 7 – Une marche aléatoire

```
rules

[] randomwalk  =>  pc(leftstrat:2,rightstrat:1,downstrat:2,upstrat:1)

[] leftstrat => first(left1,left2,left3)
[] downstrat => first(down1,down2,down3)
[] upstrat => first(up1,up2,up3)
[] rightstrat => first(right1,right2,right3)

[left1] (x,y) => S  if x > 0
        where S := (randomwalk)(x-1,y)

[left2] (x,y) => S if x == 0 and y > 0
        where S := (randomwalk)(x,y)

[left3] (x,y) => (0,0) if x == 0 and y==0


[right1] (x,y) => S if x > 0
```

```
        where S := (randomwalk)(x+1,y)

[right2] (x,y) => (0,0) if y==0

[right3] (x,y) => S
        where S := (randomwalk)(x,y)

[up1] (x,y)=> S if  y > 0
        where S := (randomwalk)(x,y+1)

[up2] (x,y) => (0,0) if x == 0

[up3] (x,y) => S
        where S :=(randomwalk)(x,y)

[down1] (x,y) => S  if y > 0
        where S := (randomwalk)(x,y-1)
[down2] (x,y) => S  if x > 0
        where S := (randomwalk)(x,y)
[down3] (x,y) => (0,0)
```

La réduction du terme `(randomwalk)(67,767)` par ELAN se décompose comme suit. `randomwalk` est un alias pour
`pc(leftstrat :2,rightstrat :1,downstrat :2,upstrat :1)`,
une stratégie est choisie par `pc` pour en fait décaler la position du vecteur soit à droite, soit à gauche, soit en haut ou en bas dans la mesure du possible (on doit respecter les effets de bord). Lorsque la nouvelle position est réécrite, l'opérateur de stratégie `(randomwalk)` est replacé devant le vecteur pour poursuivre le calcul. Cette opération n'a pas lieu si et seulement si la position était $(0,0)$ avant le pas de réécriture. On obtient le résultat $(0,0)$ en un temps moyen borné assez court, le seul qu'ELAN puisse retourner dans ce cas.

5 Conclusion et perspectives

L'implantation des stratégies de réécriture probabilistes au système ELAN permet désormais de travailler avec les systèmes probabiliste. Nous pouvons maintenant exprimer le comportement de processus et d'automates probabilisés (dont les chaînes de Markov) ainsi qu'exécuter un système de règles de réécriture correspondant au comportement d'un tel automate. A partir de ce travail nous allons pouvoir modéliser des systèmes complexes où interviennent les probabilités ainsi que vérifier de manière expérimentale certaines propriétés de la réécriture avec stratégies probabilistes.

6 Annexe

6.1 À propos de la génération de nombre aléatoire

Il n'existe pas d'algorithme déterministe permettant d'obtenir un comportement complètement aléatoire. Cependant, il existe de nombreux algorithmes permettant d'obtenir des séquences de nombres avec une très grande période et décorellés. Parmi ces algorithmes, on peut citer la classe des "LCG" pour Linear Congruencial Generator, à laquelle appartient le "random" de la librairie C. Une très bonne référence est le livre de Donald Knuth [10], où les pièges et les techniques de la génération de nombre pseudo aléatoires sont présentés. Des séquences de nombre réellement aléatoires peuvent être obtenues en observant des phénomènes physiques, tels que les désintégrations d'un échantillon de radium, ou en observant un système complexe ayant de de nombreux états interfèrant entre eux. Calculer des séquences de nombres aléatoires à partir de l'observation de d'un système très entropique est un thème de recherche aborde par l'équipe d'André Seznec à l'IRISA [12].

Références

[1] Peter Borovansky, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a Rewriting Logic Point of View. *Theoretical Computer Science*, (285) :155–185, July 2002.

[2] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http ://www.elsevier.nl/locate/entcs/volume16.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[3] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN : a functional semantics. *International Journal of Foundations of Computer Science*, 12(1) :69–98, February 2001.

[4] Olivier Bournez and Mathieu Hoyrup. Rewriting logic and probabilities. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2003.

[5] Olivier Bournez and Claude Kirchner. Probabilistic rewrite strategies : Applications to elan. In Sophie Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, July 22-24 2002.

[6] Cirstea, Horatiu and Kirchner, Claude. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3) :427–498, May 2001.

[7] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalisme SDF - reference manual. *ACM SIGPLAN Notices*, 24(11) :43–75, 1989.

[8] H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 2003.

[9] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In F. Orejas, editor, *Proceedings of 2nd CCL Workshop*, La Escala (Spain), September 1993.

[10] Donald.E Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1997.

[11] J. Meseguer. Conditional rewriting logic as unified model of concurrency. *Theoretical Computer Science*, 96(1) :73–155, 1992.

[12] André Seznec and Nicolas Sendrier. Havege : A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4) :334–346, 2003.

[13] Brand M.G.J. van den, P.E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. Technical report, CWI, 2003.