Tirages aléatoires uniformes dans des systèmes concurrents

Johan Oudinet
LRI
Bât 490 Université Paris-Sud
91405 Orsay Cedex, France
oudinet@lri.fr

Résumé

Ce rapport présente les premiers résultats expérimentaux sur la génération aléatoire uniforme de chemins dans de très grands graphes qui modélisent des systèmes concurrents, comme elle a été présentée dans [5]. L'approche se base sur des techniques de comptage et de tirage aléatoire uniforme dans des structures combinatoires et sur le fait que dans des systèmes construits à partir de composants concurrents, il est possible de combiner astucieusement des tirages uniformes dans les composants pour approcher, avec une très bonne approximation, un tirage uniforme dans le système global. Ce rapport décrit les implémentations des méthodes proposées dans [5], analyse les résultats d'une suite de tests de performance, et généralise le cas où les composants concurrents se synchronisent à plusieurs symboles et plusieurs occurrences de synchronisations.

MotsClefs

génération uniforme, modèles concurrents, modèles modulaires, model-checking, test basé sur des modèles.

1. INTRODUCTION

Les marches aléatoires sont couramment utilisées en simulation de logiciel, test structurel, test basé sur des modèles, et plus récemment en *model checking*. En général, les marches aléatoires classiques (choix au hasard du prochain noeud parmi les successeurs) ne permettent pas de garantir une bonne couverture du graphe sous-jacent.

Dans [4, 11], Denise, Gaudel et Gouraud montrent comment utiliser des méthodes de dénombrement et de tirage uniforme dans des structures combinatoires pour tirer des chemins soit uniformément, soit biaisés par un critère de couverture donné. L'outil AuGuSTe [10] était une première application de cette approche pour faire du test statistique structurel ; des expériences sur des programmes C ont été menées avec des graphes de contrôle allant jusqu'à 100 noeuds.

Pour traiter des modèles encore plus grands, comme ceux modélisés par des systèmes concurrents, Denise, Gaudel, Gouraud, Lassaigne et Peyronnet ont eu l'idée de raisonner sur les modèles concurrents, plus petits, qui composent le très grand modèle. Dans [5], ils estiment le nombre de chemins (de longueur fixée) sans construire le modèle global, mais en utilisant des techniques issues de l'analyse combinatoire [6, 12]. Les résultats permettent de faire de la génération uniforme de chemins dans des graphes très grands, avec une très bonne approximation.

Dans ce rapport, nous commencerons par un rappel sur le test et les marches aléatoires, avant d'explorer les limites de la méthode utilisée dans AuGuSTe pour le tirage uniforme de chemins dans un seul graphe.

Dans la section 3, nous décrirons notre implémentation et nous analyserons les résultats expérimentaux sur des graphes de taille moyenne : jusqu'à 18746 noeuds pour des chemins de longueur 2000.

Puis, la section 4 présentera l'implémentation du tirage uniforme approché dans de très grands graphes qui sont le produit asynchrone de modèles concurrents et de taille moyenne comme ceux considérés à la section 3. Des chemins de longueur 8000 ont été obtenus sur des modèles contenant 4.10^{24} états.

Ensuite, la section 5 mettra en pratique les résultats précédents pour faire du tirage uniforme dans un système de modèles concurrents avec la présence de synchronisations. Les résultats expérimentaux sont limités à une seule synchronisation par modèle. Des chemins de longueur 500 ont été obtenus sur des modèles de taille équivalente à ceux utilisés dans le cas asynchrone.

Enfin, nous étudierons le cas général où les modèles se synchronisent sur différents symboles et avec plusieurs occurrences de ces symboles par module à la section 6. Le cas de la synchronisation partielle, où certains modules évoluent indépendamment pendant que d'autres se synchronisent, est le cas qui pose le plus de difficultés ; un début de solution est proposé à la section 7.

2. CONTEXTE

Cette section est consacrée à la définition des termes qui seront largement utilisés tout au long de ce rapport. Le lecteur habitué au test de logiciel, au model-checking et aux notions de marches aléatoires et de parcours dans un graphe est invité à se rendre directement à la section suivante.

2.1 Le test de logiciel

Le test de logiciel est un procédé visant à vérifier si le comportement d'un système est conforme aux spécifications établies par ses concepteurs. La norme IEEE 829 définit le test comme tel :
Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification.

On notera au passage que le test n'essaye pas de prouver la correction du système, qui est un problème indécidable, mais uniquement de détecter des anomalies. Ainsi, Edsger Dijkstra disait : "*Le test de programmes peut être utilisé pour détecter la présence de bogues, mais jamais leur absence.*"

Le test est une méthode dynamique d'évaluation d'un système, par opposition aux méthodes statiques qui ne nécessitent pas l'exécution du logiciel. En effet, le test consiste à lancer une version exécutable du code du futur système, puis observer et évaluer les résultats. Si c'est un modèle préliminaire qui est exécuté (un prototype, un modèle de conception enrichi pour être exécutable, ou une spécification formelle) alors on parle de *simulation* au lieu de *test* qui est réservé à l'évaluation du système final.

Les activités du test peuvent être regroupées en 4 étapes :

1. la sélection des tests,
2. la soumission des tests sélectionnés,
3. décider du succès ou de l'échec des tests soumis,
4. évaluer la qualité des tests effectués.

Ces activités posent de nombreux problèmes. La sélection d'un *jeu de tests* doit choisir judicieusement un sous-ensemble fini des entrées possibles du système ; ce choix est généralement dépendant d'un critère de sélection à atteindre. Ensuite, l'étape de soumission des tests nécessite souvent l'utilisation ou le développement d'un environnement de test et la simulation des modules manquants par l'ajout de modules fictifs. Une fois exécuté, décider si un test a échoué ou non fait intervenir un *oracle* qui peut être très compliqué à construire : l'oracle est parfois plus complexe que le logiciel sous test ! Enfin l'étape d'évaluation de la qualité du jeu de test est primordiale pour décider de l'arrêt du test. En plus des nombreux problèmes que posent ces étapes séparément, il est nécessaire de les aborder ensemble car cela n'aurait guère de sens de sélectionner un test qu'on ne pourrait pas mettre en oeuvre ou dont le résultat est inconnu.

2.1.1 Sélection des tests

Cette étape clef, comme nous venons de le voir, est celle qui nous intéresse particulièrement car les méthodes développées dans ce rapport pourront servir à améliorer cette sélection.

De nombreuses solutions ont déjà été proposées pour la sélection des tests. On ne cherche pas à donner une énumération exhaustive mais seulement une classification habituelle qui consiste à distinguer les méthodes basées sur :

1. le domaine d'entrée du programme,
2. la structure du programme,
3. des spécifications plus ou moins formelles du programmes,
4. les modèles de fautes.

Certaines méthodes sont mixtes, c'est-à-dire qu'elles relèvent de plusieurs catégories, et d'autres sont difficilement classables.

2.1.1.1 Sélection basée sur le domaine d'entrée

Le domaine d'entrée d'un programme est en général infini, ou trop large pour envisager un test exhaustif. Il existe deux grandes approches pour se ramener à un jeu de tests fini : le test aléatoire et le test par partition.

Le test aléatoire consiste à définir une distribution de probabilité sur l'ensemble des entrées et à effectuer des tirages aléatoires avec cette distribution. Malheureusement, l'utilisation d'une distribution uniforme, bien que simple à mettre en oeuvre, est généralement décevante car il s'avère que dans la plupart des cas un gros sous-ensemble du domaine correspond au cas *normal* d'exécution du programme et seul un petit nombre d'entrées conduit à des cas particuliers d'exécution du programme qui ne sont donc pas couverts en utilisant une distribution uniforme. Une distribution basée sur un *profil d'usage* [15, chap. 5] permet au moins de détecter les erreurs qui ont le plus de chances de se produire lorsque le système sera en opération. Mais la difficulté se situe dans la création d'un profil d'usage qui soit suffisamment fidèle au client futur ; ce qui n'est pas possible pour tous les systèmes.

Le test par partition consiste à décomposer le domaine d'entrée en un nombre fini de partitions sous l'hypothèse que toutes les valeurs d'une partition conduisent au même verdict. Il suffit alors de sélectionner une valeur par partition. La décomposition peut se faire directement en fonction du type de données : pour des entiers on choisi une valeur négative, zéro et une valeur positive, ou être guidé par le domaine d'application, la spécification ou le programme. On rajoute souvent, en plus d'une valeur par partition, des valeurs aux limites de chaque partition.

2.1.1.2 Sélection basée sur la structure

C'est certainement le domaine en test qui a été le plus étudié. Ces techniques de sélection, appelées *tests structurels*, se basent sur une représentation graphique du programme : la plus connue étant le graphe de contrôle. Il s'agit d'un graphe dont :

- chaque sommet désigne soit un bloc élémentaire de programme, soit un prédicat d'une instruction conditionnelle ou d'une boucle,
- chaque arc indique un enchaînement possible entre deux sommets.

Un bloc élémentaire contient une suite d'instructions indissociables : si on exécute la première instruction d'un bloc élémentaire alors les suivantes seront forcément exécutées.

La figure 1 représente le graphe de contrôle de l'algorithme 1.

Ainsi, chaque exécution du programme correspond à un chemin dans le graphe de contrôle. La réciproque est généralement

Algorithme 1 Algorithme d'Euclide

Entrées: $a, b \in \mathbb{N}$
Sorties: le pgcd de a et b
  $x \leftarrow \max(a, b)$
  $y \leftarrow \min(a, b)$
  tantque $y \neq 0$ **faire**
    $x \leftarrow y$
    $y \leftarrow x \mod y$
  fin tantque
  Retourner x.

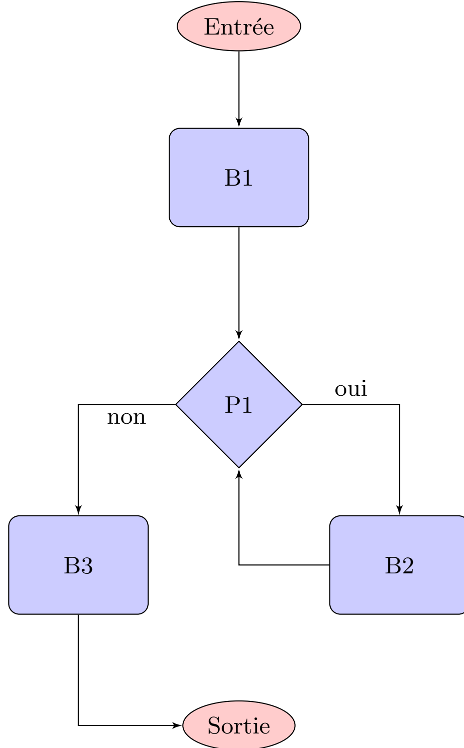

Figure 1 : Le graphe de contrôle de l'algorithme d'Euclide.

fausse car il existe parfois des *chemins infaisables*, c'est-à-dire qu'il est impossible d'obtenir une exécution du programme qui passe par ce chemin.

Les critères de test structurel les plus classiques sont :
– la couverture des instructions : on est passé au moins une fois dans chaque bloc élémentaire après l'exécution du jeu de tests,
– la couverture des enchaînements : passer au moins une fois par chaque arc du graphe,
– la couverture de tous les chemins, qui est généralement impossible (si il y a une boucle dans le programme) ou trop coûteuse et on se ramène donc à un critère plus faible comme la couverture de tous les chemins qui passe au maximum i fois dans chaque boucle.

Par exemple le chemin $\{E, B1, P1, B2, B3, S\}$ permet de couvrir à la fois toutes les instructions et tous les enchaînements. Pour couvrir tous les chemins qui passent au plus 1 fois dans chaque boucle, il faut aussi rajouter le chemin $\{E, B1, P1, B3, S\}$ au jeu de tests. Et pour le critère tous les chemins qui passent au plus 2 fois dans chaque boucle, on aura 3 chemins : les 2 précédents plus le chemin $\{E, B1, P1, B2, B2, B3, S\}$.

Pour obtenir un test à partir d'un chemin il faut d'abord construire le prédicat associé au chemin grâce à une analyse statique qui s'appelle l'évaluation symbolique. On obtient alors un *cas de test*. Il faut ensuite résoudre ce *cas de test* à l'aide d'un solveur de contraintes afin de déterminer quelles entrées du programme permettent d'exécuter ce chemin. Savoir si un chemin est satisfiable ou non est un problème indécidable dans le cas général. Et se ramener aux cas particuliers décidables écarterait la plupart des langages de programmation. Néanmoins, des progrès récents dans les outils de résolution de contraintes (techniques de *randomisation* et utilisations d'heuristiques) ont permis d'améliorer la situation en pratique, en réduisant les cas où le solveur ne trouve pas de solution à un prédicat satisfiable.

2.1.1.3 Sélection basée sur une spécification

On test un programme non seulement pour chercher les fautes qu'il contient mais aussi pour vérifier que le programme possède toutes les fonctionnalités qu'il est censé remplir. De plus, quand on achète ou sous-traite un logiciel, on n'a pas forcément accès à sa structure interne ; lorsque le testeur ne peut observer la structure interne, il fait du test *boîte noire*. Pour toutes ces raisons, l'utilisation de la spécification pour construire le jeu de tests est indispensable. Ce type de test s'appelle du *test fonctionnel*.

Les méthodes de sélection sont dépendantes du format sous lequel est disponible la spécification. Dans les années 70, les spécifications étaient écrites en langages naturels et des tables de correspondance entre les entrées et les sorties attendues étaient manuellement extraites à partir de ces descriptions. Puis, l'idée d'utiliser des langages plus formels pour décrire ces spécifications sous forme d'automates a émergé. Sous l'influence de ce qui se faisait dans le domaine des circuits logiques, des tests ont été générés à partir de machines à états finis (FSM) [14]. Et, après l'apparition de langages de spécification formelle mieux adaptés au logiciel (LOTOS permet de décrire des protocoles de communications à l'aide de systèmes de transitions étiquetées (LTS) [2]), de nombreuses méthodes de test fonctionnel ont été proposées.

2.1.1.4 Sélection basée sur les modèles de fautes

Le *test par mutation* a été défini à l'origine comme une méthode de sélection de tests. Mais il s'est avéré très utile pour évaluer l'efficacité des jeux de tests.

Le test par mutation consiste à créer des mutants du programme en lui rajoutant des fautes : chaque mutant est identique au programme à une différence près, par exemple en remplaçant un $<$ par un $\leq$ dans une condition de boucle. On obtient alors un ensemble de mutants contenant chacun une faute.

On évalue ensuite la qualité d'un jeu de test en mesurant le pourcentage de mutants qu'il détecte. Ce type d'évaluation est très utilisé pour comparer l'efficacité des méthodes de test.

2.2 Le model-checking

Le model-checking consiste à vérifier si un modèle donné satisfait une certaine formule. Le modèle est représenté par un automate ou un système de transitions et il décrit les futurs comportements du système à développer. Le model-checking peut être utilisé en aval du test pour des vérifications préliminaires ou comme outil : soit de test fonctionnel pour générer des tests à partir du modèle, soit de test structurel (on parle alors de *software model-checking*).

La formule est décrite dans une logique temporelle (LTL ou CTL). Ce qui permet d'exprimer des propriétés d'atteignabilité (absence de blocage), sûreté (non occurrence de certains événements), vivacité ou encore équité.

Il existe des implémentations libres et très puissantes qui permettent de vérifier ou de réfuter automatiquement ces propriétés sur des modèles de très grande taille. On peut maintenant traiter des systèmes de plusieurs millions d'états alors qu'on en était à 50000 états il y a seulement quelques années. Néanmoins, lorsqu'on traite des logiciels complexes, les modèles restent souvent trop grands malgré l'utilisation de simplifications et d'abstractions.

2.3 Le problème de l'explosion combinatoire

Dans la plupart des problèmes réels, les disciplines qui se basent sur un modèle du système (model-checking, test structurel, test fonctionnel) sont confrontées à une explosion combinatoire du nombre d'états de ces modèles. Il devient très difficile d'explorer le graphe sous-jacent au modèle de manière exhaustive.

Il existe deux approches orthogonales pour pallier cette explosion combinatoire : réduire la taille du modèle à l'aide d'hypothèses d'abstractions, et ne pas parcourir tout le graphe en utilisant par exemple des marches aléatoires.

2.4 Les marches aléatoires

Les marches aléatoires sont couramment utilisées car elles ne nécessitent qu'une connaissance locale du graphe : uniquement l'état courant et l'ensemble de ses successeurs. Ce qui permet l'exploration de graphe gigantesque sans aucun soucis. L'algorithme classique de marches aléatoires est le suivant :

1. prendre en entrées un graphe et une longueur n,
2. générer un chemin aléatoire w de longueur n : $w = (s_0, \ldots, s_n)$ tel qu'à chaque étape, on a choisi s_{i+1} uniformément parmi les successeurs de s_i,
3. soumettre w à l'implémentation qui est évaluée afin d'essayer de détecter une anomalie.

L'inconvénient majeur de cette approche est que nous ne connaissons pas la distribution de probabilité induite sur les chemins du modèle, car elle dépend de la topologie du graphe sous-jacent. Par exemple, on voit que si nous effectuons une marche aléatoire sur le graphe de la figure 2, alors le chemin $\{a, c, f\}$ a 4 fois plus de chances d'apparaître que le chemin $\{b, e, j\}$. En effet :

$$P(\{a,c,f\}) = \frac{1}{2}$$
$$P(\{b,e,j\}) = \frac{1}{8}$$

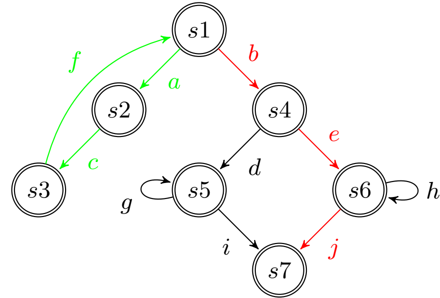

Figure 2 : Exemple d'un graphe représentant un modèle à 7 états, dont 2 chemins de longueur 3 sont représentés en vert et en rouge : $\{a, c, f\}$ et $\{b, e, j\}$.

2.5 Objectifs et limites du stage

Ce rapport se focalise sur le tirage uniforme de chemins dans de très grands modèles.

On considère des LTS finis définis comme tel : Un LTS est une structure $M = (S, T, s_0, X, F)$ où S est un ensemble d'états, s_0 est l'état initial, $T \subseteq S \times X \times S$ est une relation de transitions, X est un ensemble d'actions et $F \subseteq S$ désigne l'ensemble des états finaux. Le lecteur souhaitant approfondir ses connaissances dans le domaine des systèmes de transitions pourra se référer à l'ouvrage d'Arnold [1].

On note $|M|$ la taille de M (i.e., son nombre d'états). Avec cette définition, nos LTS peuvent être non-déterministes : un état peut avoir plusieurs transitions sortantes étiquetées par la même action et allant vers des états différents. On considère souvent tous les états d'un LTS comme des états terminaux, ici nous voulons aussi modéliser les programmes où seuls certains états sont finaux : l'ensemble F permet de définir ces deux cas.

Une séquence d'états est appelée un *chemin* et une *trace* est la séquence des actions empruntées par un chemin.

Finalement, un LTS n'est rien d'autre qu'un graphe avec une sémantique particulière et dans la suite de ce rapport nous confondrons allégrement le LTS avec son graphe sous-jacent. Ainsi, nous utiliserons régulièrement le graphe sous-jacent au modèle et une action sera alors un symbole qui étiquette une transition de ce graphe.

Comme vu à la section 2.1.1.2, plusieurs critères de test peuvent être considérés. Nous nous intéresserons ici à la couverture de tous les chemins de longueur n. En fait, grâce à une petite astuce qui consiste à rajouter une transition fictive sur l'état initial, ce critère permet aussi de couvrir tous les chemins de longueur inférieure ou égale à n [4].

En pratique, les modèles sont souvent trop grands pour pouvoir être étudiés dans leur globalité. C'est pourquoi nous

cherchons à guider la marche aléatoire en biaisant le choix des successeurs de manière à garantir l'équiprobabilité des chemins de longueur n du modèle. Ainsi nous gardons les avantages des marches aléatoire, à savoir pouvoir explorer des graphes gigantesques, tout en conservant la garantie d'avoir une couverture uniforme de tous les chemins.

Seul le chemin de longueur n généré par la marche aléatoire nous intéresse, ce qui fait qu'en réalité nous ne raisonnons pas en terme de marche aléatoire mais plutôt en terme de tirage d'un chemin parmi tous les chemins de longueur n.

Les sections 3 à 5 expliquent les implémentations que j'ai réalisées des solutions proposées dans [5] puis nous verrons un nouvel algorithme qui généralise ces solutions lorsque les modèles concurrents se synchronisent sur plusieurs actions.

3. TIRAGE EXACT DANS UN MODÈLE

Dans cette section, l'objectif est d'explorer les limites de la méthode utilisée dans AuGuSTe pour le tirage uniforme de chemins dans un seul graphe. J'ai créé, dans ce but, une routine permettant de générer des chemins uniformément à partir du graphe d'un modèle au format BCG (Binary Coded Graphs [9]), qui est un format de représentation de graphes développé par l'équipe VASY à l'INRIA Rhône-Alpes dans le cadre de la boite à outils pour la modélisation de systèmes distribués et de protocoles de communication (CADP [8]). Il s'agit de tirer des chemins de longueur fixée parmi tous les chemins de même longueur du graphe d'un modèle.

3.1 Principe

Avant de simuler une marche aléatoire uniforme sur les chemins d'une longueur fixée : n, il est nécessaire de connaître le nombre de chemins d'une longueur inférieure ou égale à n partant de chacun des sommets du graphe. En construisant la table 1, on peut ensuite générer uniformément des chemins de longueur 3. Ainsi, tous les chemins de longueur 3 ont une probabilité de $\frac{1}{5}$:

$$\begin{aligned} P(\{b,e,j\}) &= \frac{s4(2)}{s1(3)} \times \frac{s6(1)}{s4(2)} \times \frac{s7(0)}{s6(1)} \\ &= \frac{2}{5} \times \frac{1}{2} \times 1 \\ &= \frac{1}{5} \end{aligned}$$

où $sX(n)$ représente le nombre de chemins de longueur n partant de l'état sX.

Ces tables de comptage sont calculées à l'aide de la relation de récurrence suivante :

$$\begin{cases} sX(0) = 1 & \text{si } sX \in F \\ sX(0) = 0 & \text{sinon} \\ sX(n) = \sum_{s \to sY} sY(n-1) & \end{cases}$$

Pour calculer ces tables de comptage et générer des chemins de longueur n, j'ai utilisé le logiciel de calcul formel MuPAD [3] avec la bibliothèque MuPAD-Combinat [18].

Table 1 : Nombre de chemins de longueur inférieure ou égale à 3 partant d'un sommet précis du modèle de la figure 2.

	0	1	2	3
s1	1	2	4	5
s2	1	1	1	2
s3	1	1	2	4
s4	1	2	4	4
s5	1	2	2	2
s6	1	2	2	2
s7	1	0	0	0

Cette bibliothèque permet de générer un programme C effectuant des tirages uniformes parmi les chemins de taille n que contient une structure combinatoire.

La première étape de mon travail a consisté à extraire la structure combinatoire associée à la description du modèle au format BCG. Par exemple, la structure combinatoire associée au modèle de la figure 2 est représentée dans la table 2. Un programme C++ a été écrit afin d'obtenir un script MuPAD contenant la structure combinatoire associée au modèle. Ce script génère un programme C effectuant des tirages uniformes parmi les chemins de taille n que contient la structure combinatoire.

Table 2 : Structure combinatoire du modèle de la figure 2.

$$\begin{aligned} \{\ s1 &= \epsilon + a.s2 + b.s4 \\ s2 &= \epsilon + c.s3 \\ s3 &= \epsilon + f.s1 \\ s4 &= \epsilon + d.s5 + e.s6 \\ s5 &= \epsilon + g.s5 + i.s7 \\ s6 &= \epsilon + h.s6 + j.s7 \\ s7 &= \epsilon\ \} \end{aligned}$$

Une fois compilé, le programme prend en paramètres la longueur des chemins à générer et le nombre de chemins souhaités. Si on indique 0 pour le nombre de chemins à générer, alors le programme renvoie le nombre de chemins de la longueur fixée que contient la structure combinatoire.

3.2 Mesures et résultats

Les mesures ont été réalisées sur le serveur de l'équipe asspro (Intel 2.8GHz avec 1GB de mémoire vive). Les graphes au format BCG proviennent de la base de modèles VLTS (Very Large Transition Systems [7]). Ces modèles sont des cas correspondants à des systèmes industriels réels.

La table 3 mesure le temps mis par le script MuPAD-Combinat pour générer les programmes C permettant d'effectuer des tirages uniformes de chemins. Le temps d'exécution est au pire quadratique en la taille du modèle (il est linéaire pour

les graphes qui ont un degré sortant borné), mais elle n'a besoin d'être effectuée qu'une seule fois avant de tirer plusieurs chemins.

Table 3 : Temps mis par le script MuPAD pour générer le programme C de tirage uniforme.

	# états	# transitions	MuPAD → C
vasy_0_1	289	1224	2s
cwi_1_2	1952	2387	32s
vasy_1_4	1183	4464	20s
cwi_3_14	3996	14552	85s
vasy_5_9	5486	9676	10m
vasy_8_24	8879	24411	43m
vasy_8_38	8921	38424	17m
vasy_10_56	10849	56156	34m
vasy_18_73	18746	73043	4h

Le programme C généré peut tirer des chemins de longueurs différentes. Il doit donc construire la table de comptage (du nombre de chemins de longueur i pour tout $0 \leq i \leq n$ partant de chaque état) avant de tirer un ou plusieurs chemins. La complexité arithmétique pour construire cette table de comptage est $O(n \times |G|)$ où $|G|$ est la taille du graphe [6]. La table 4 montre les temps de construction de ces tables de comptage. Le symbole ∞ signifie qu'il n'y avait pas assez de mémoire pour la table.

Table 4 : Temps mis pour construire la table de comptage. Note : il n'y a aucun chemin de longueur supérieure à 61 dans le modèle cwi_3_14, d'où la construction rapide de ses tables.

	200	1000	2000	3000	5000	8000
vasy_0_1	0.1s	0.3s	0.9s	1.5s	3.7s	8.8s
cwi_1_2	0.2s	1.5s	3.5s	5.8s	11.2s	21.5s
vasy_1_4	0.2s	1.1s	2.8s	5.1s	11.9s	∞
cwi_3_14	0.2s	0.8s	1.4s	2.1s	3.5s	5.5s
vasy_5_9	0.3s	4.9s	12.2s	22.0s	∞	∞
vasy_8_24	1.0s	9.0s	22.0s	∞	∞	∞
vasy_8_38	0.6s	4.4s	11.4s	22.9s	∞	∞
vasy_10_56	3.2s	22.5s	∞	∞	∞	∞
vasy_18_73	2.7s	20.0s	50.2s	∞	∞	∞

Enfin, une fois la table de comptage créée, la génération de plusieurs chemins de longueur n peut s'effectuer en $O(n)$ opérations arithmétiques. La table 5 mesure le temps mis pour générer 100 chemins de longueur n (n variant entre 200 et 8000) sans prendre en compte le temps de prétraitement pour la construction des tables de comptage. En sachant que le programme C commence par construire les tables de comptage avant de générer des chemins de longueur n, le nombre affiché dans le tableau est le résultat de l'opération suivante : $T(100)-T(0)$ où $T(100)$ désigne le temps mis pour générer 100 chemins de longueur n (avec la construction des tables de comptage), et $T(0)$ le temps pour compter le nombre de chemins de longueur n (c'est-à-dire juste la construction des tables de comptage).

Ces tables montrent l'efficacité de notre approche : l'étape de prétraitement peut être assez lente - 4h pour le plus grand

Table 5 : Temps mis pour générer 100 chemins.

	200	1000	2000	3000	5000	8000
vasy_0_1	0.0s	0.9s	2.9s	6.3s	15.9s	40.1s
cwi_1_2	0.1s	0.6s	1.6s	3.1s	7.5s	16.0s
vasy_1_4	0.1s	1.0s	3.2s	6.7s	18.2s	∞
cwi_3_14	0.0s	0.0s	0.0s	0.0s	0.0s	0.1s
vasy_5_9	0.0s	0.9s	2.4s	5.2s	∞	∞
vasy_8_24	0.2s	0.8s	2.4s	∞	∞	∞
vasy_8_38	0.1s	1.6s	5.5s	10.8s	∞	∞
vasy_10_56	0.0s	1.3s	∞	∞	∞	∞
vasy_18_73	0.2s	0.9s	3.3s	∞	∞	∞

modèle testé qui avait 18746 états - mais elle n'est faite qu'une seule fois, et l'étape de génération est extrêmement rapide. Ce qui permet de générer beaucoup de chemins et ainsi de réaliser une exploration intensive du graphe. Cependant, ces tables montrent aussi les limites de cette approche : il est nécessaire de garder en mémoire des tables de comptages d'une taille importante. Et même si beaucoup de mémoire était disponible, le temps de génération étant linéaire en la taille du graphe, gérer des graphes vraiment très grands resterait illusoire.

4. TIRAGE APPROCHÉ DANS DES MODÈLES CONCURRENTS ASYNCHRONES

Dans la section précédente, on tirait uniformément des chemins dans des graphes de taille moyenne. Dans cette section, on considère des graphes beaucoup plus grands, qui sont le résultat de la composition asynchrone de r modules (M_1, ..., M_r). À chaque module correspond un graphe de taille moyenne. La composition asynchrone est le produit de r modèles. Un chemin dans le modèle global est le mélange de chemins tirés dans chacun des r modèles. J'ai implémenté la solution proposée dans [5] dans le but de manipuler des graphes très grands qui correspondent à des systèmes concurrents. L'idée principale est d'éviter de construire le modèle global en combinant astucieusement des tirages uniformes de chemins locaux à chaque module pour obtenir une très bonne approximation d'un tirage uniforme de chemins dans le système global.

Le cas de la composition asynchrone est une première étape : l'algorithme présenté dans cette section sera utile pour la composition synchrone.

4.1 Principe

La génération uniforme à partir de la composition asynchrone de r modules peut se résumer en 4 étapes successives :

1. fixer la longueur n d'un chemin,
2. choisir les longueurs $(n_1, \ldots, n_r)$ des chemins, qui seront tirés dans chacun des r modules, avec une probabilité appropriée,
3. pour chaque module M_i, tirer un chemin de longueur n_i en utilisant l'algorithme vu à la section 3),

4. mélanger les r chemins pour obtenir un chemin de longueur n.

C'est l'item 2 qui est l'étape la plus compliquée (on verra un algorithme trivial pour mélanger r chemins en garantissant l'uniformité).

Le r-uplet $(n_1, \ldots, n_r)$ tel que $n_1 + \cdots + n_r = n$, doit être choisi avec la probabilité :

$$P(n_1, \ldots, n_r) = \frac{\binom{n}{n_1 \cdots n_r} l_1(n_1) \cdots l_r(n_r)}{l(n)}$$

où $l_i(k)$ est le nombre de chemins de longueur k dans le module M_i et $l(n)$ est le nombre de chemins de longueur n dans le modèle global. Le problème étant que $l(n)$ est trop coûteux à calculer car il nécessite le modèle global. Néanmoins, il est possible de l'estimer en utilisant un théorème dans [6] qui permet, sous certaines conditions qui sont généralement vérifiées en pratique, d'approximer le nombre de chemins de longueur n par : $l(n) \sim C\omega^n$ avec C et ω qui sont des constantes calculées à partir de la fonction génératrice associée au modèle global (nous verrons comment obtenir ces valeurs un peu plus loin dans cette section). Ces constantes sont trop coûteuses à calculer directement sur le modèle global, cependant, en utilisant celles correspondantes aux r modules (i.e., $\omega_1, \ldots, \omega_r$ et $C_1, \ldots, C_r$) plus le fait que le modèle global soit le résultat de la composition de ces r modules, on obtient l'approximation suivante :

$$l(n) \sim (C_1 \ldots C_r)(\omega^1 + \cdots + \omega_r)^n$$

Cette fois ci, l'approximation obtenue permet de calculer $l(n)$ sans construire le modèle global. La probabilité du r-uplet $(n_1, \ldots, n_r)$ devient :

$$P(n_1, \ldots, n_r) \sim \frac{\binom{n}{n_1 \cdots n_r} \omega_1^{n_1} \cdots \omega_r^{n_r}}{(\omega_1 + \cdots + \omega_r)^n}$$

Nous allons maintenant détailler les différentes étapes de l'algorithme de génération uniforme de chemins dans un système concurrent asynchrone depuis le calcul des ω_i jusqu'au mélange des r chemins.

4.1.1 Calcul des ω_i

Un programme en MuPAD-Combinat a été réalisé afin de calculer le ω d'un module à partir de sa description au format bcg. Cette valeur correspond à l'inverse du pôle réel de plus petite valeur absolue du dénominateur de la fonction génératrice associée au module. C'est-à-dire que pour obtenir ω il faut tout d'abord trouver la fonction génératrice, qui est la solution à un système linéaire défini à partir du graphe du module. Une fois cette fonction génératrice trouvée, qui est forcément une fonction rationnelle car le système est linéaire, il faut chercher les racines du dénominateur de cette fonction. Enfin, ω est égale à l'inverse de la racine réelle de plus petite valeur absolue.

4.1.1.1 Construction et résolution d'un système linéaire

Le graphe (au format bcg) représentant le module est utilisé pour établir un système linéaire avec autant d'équations que

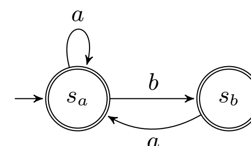

Figure 3 : Le module M_1 génère des traces appartenant au langage $(a|ba) * (\epsilon|b)$

de noeuds dans le graphe. Par exemple, le système correspondant au module M_1 (figure 3) est :

$$\begin{cases} s_a(z) = 1 + z \cdot s_a(z) + z \cdot s_b(z) \\ s_b(z) = 1 + z \cdot s_a(z) \end{cases}$$

Deux approches différentes peuvent être ensuite utilisées pour résoudre ce système linéaire [19, chap. 2]

- les méthodes *directes* qui résolvent l'équation $A \cdot x = b$ en $O(m^3)$ où m désigne l'ordre de la matrice A (décomposition LU, élimination de Gauss, élimination de Gauss-Jordan). L'algorithme de Strassen permet de résoudre la même équation en $O(m^{2.76})$ et celui de Coppersmith et Winograd en $O(m^{2.38})$ mais les constantes cachées sont prohibitives,
- les méthodes dites *itératives*, qui dans le cas où A est une matrice creuse peuvent résoudre l'équation en $O(m^2)$ (cf. l'algorithme du gradient biconjugué) à condition d'avoir une bonne représentation de la matrice (qui est dépendante des motifs des valeurs non nulles. Par exemple, une matrice tribande).

Pour l'instant, c'est la fonction *linsolve* de MuPAD (dans le package *numeric*) qui est utilisée en mode *Symbolic* (elle effectue une élimination de Gauss avec un pivot partiel).

Mais les matrices que nous avons en pratique sont des matrices creuses (car le degré sortant de chaque noeud est souvent borné par une constante très inférieure au nombre de noeud). Il serait probablement plus efficace d'utiliser une des méthodes dites *itératives*.

4.1.1.2 Extraction de ω

La solution au système linéaire précédent est une fonction rationnelle :

$$f(z) = \frac{N(z)}{D(z)}$$

Par exemple, pour M_1, on obtient la fonction f_1 suivante :

$$f_1(z) = \frac{1 + z}{1 - z - z^2}$$

Il nous faut ensuite un algorithme pour extraire les racines du polynôme $D(z)$. L'algorithme de Laguerre est une méthode itérative qui permet de trouver numériquement les racines d'un polynôme. Cet algorithme a une convergence cubique, ce qui fait qu'il est plus performant que l'algorithme de Newton qui lui a une convergence quadratique.

En pratique, nous utilisons la fonction *polyroots* de MuPAD (toujours dans le package *numeric*) qui trouve toutes les

racines d'un polynôme en utilisant la méthode itérative de Laguerre.

Ensuite, comme seule la racine réelle qui a la plus petite valeur absolue nous intéresse, on la récupère :

$$\gamma_i \quad tq \quad |\gamma_i| < |\gamma_j| \quad \forall j \neq i$$

Si on suppose que le graphe représentant le modèle est apériodique et fortement connexe (ce qui est généralement vérifié en pratique [5]), alors cette racine réelle est unique et son inverse est égal à :

$$\omega = \frac{1}{\gamma_i}$$

En gardant l'exemple de M_1, on obtiendrait $\omega_1 \sim 0.618$

4.1.2 *Tirage d'un r-uplet* $(n_1, \ldots, n_r)$

Un autre programme C++ a été réalisé pour effectuer le tirage d'un r-uplet $(n_1, \ldots, n_r)$ tel que $n_1 + \cdots + n_r = n$, correspondant aux longueurs des chemins qui devront être générés dans chacun des r modules, à partir du r-uplet $(\omega_1, \ldots, \omega_r)$.

Le langage C++ a été choisi pour cette tâche, car il existe une bibliothèque de générateurs de nombres aléatoires uniformes et performants écrite dans ce langage : Boost.Random [16].

Le *mt19937* est le générateur que j'ai retenu, car c'est celui qui est conseillé par Boost.Random. Il s'agit d'un générateur de Mersenne Twister (une congruence linéaire avec le *twister*, qui est une équation supplémentaire permettant de rendre plus aléatoire la séquence de nombres). Il a un cycle de $2^{19937}-1$ et permet de générer plus de 6 millions de nombres par seconde. Le *mt19937* est donc rapide tout en ayant une qualité acceptable (voir [16] pour une comparaison des performances et des qualités des différents générateurs de Boost.Random).

L'algorithme consiste à reproduire n fois le tirage d'un n_i avec probabilité :

$$P(n_i) = \frac{\omega_i}{\sum_{j=1}^{r} \omega_j}.$$

Ainsi, le r-uplet $(n_1, \ldots, n_r)$ obtenu aura bien une probabilité de :

$$P(n_1, \ldots, n_r) = \frac{\binom{n}{n_1 \cdots n_r} \omega_1^{n_1} \cdots \omega_r^{n_r}}{(\omega_1 + \cdots + \omega_r)^n}$$

4.1.3 *Tirage uniforme de chemins de longueur* n_i *dans les modules* M_i

On utilise pour cela les versions compilées des programmes C, décrit à la section 3, permettant de générer uniformément des chemins de longueur fixée dans un modèle. On a donc un programme par module, et le programme correspondant au module M_i est utilisé pour générer un chemin w_i de longueur n_i.

Table 6 : Description pour chaque test de la taille du modèle global ainsi que le nombre de modules qui le composent. En sachant qu'ici seul le modèle vasy_0_1 a été utilisé comme module.

nom	# états	# modules
test 2	$8 \cdot 10^4$	2
test 3	$2.4 \cdot 10^7$	3
test 4	$7 \cdot 10^9$	4
test 5	$2 \cdot 10^{12}$	5
test 6	$5.8 \cdot 10^{14}$	6
test 7	$1.7 \cdot 10^{17}$	7
test 10	$4 \cdot 10^{24}$	10

Table 7 : Temps mis pour le *prétraitement* (toutes les opérations qui sont faites une seule fois avant de générer un ou plusieurs chemins).

	200	1000	2000	3000	5000	8000
test 2	33.5s	34.3s	34.6s	35.3s	36.5s	39.4s
test 3	50.9s	51.3s	51.4s	52.1s	53.0s	55.3s
test 4	68.1s	68.0s	68.7s	69.3s	69.9s	71.7s
test 5	85.3s	85.2s	85.2s	86.8s	86.8s	88.6s
test 6	94.1s	98.3s	98.3s	99.4s	1m40	1m40
test 7	1m54	1m56	1m54	1m56	1m57	1m57
test 10	2m36	2m36	2m36	2m36	2m37	2m39

4.1.4 *Mélange des r chemins*

Enfin, un programme C++ à été réalisé pour mélanger uniformément les r chemins : $w_1, \ldots, w_r$. Ce programme prend en entrée r chemins et renvoie un chemin w de longueur n (car la somme des tailles des r chemins est égale à n). L'algorithme est le suivant :

1. $w = \epsilon$, $n_i = |w_i|$ $\forall i \in [1..r]$, $n = \sum_{i=1}^{r} n_i$
2. choisir i avec la probabilité :
$$P(i) = \frac{n_i}{n},$$
3. $w = w + f(w_i)$, avec $f(w_i)$ qui extrait la première lettre de w_i,
4. $n = n - 1$ et $n_i = n_i - 1$
5. répéter les 3 opérations précédentes n fois. Le résultat est w.

4.2 Mesures et résultats

La table 6 décrit les différents modèles qui ont été utilisés pour la génération de chemins dans des systèmes concurrents asynchrones.

Ensuite, la table 7 résume le temps global mis pour la phase de *prétraitement*, c'est-à-dire toutes les étapes qui peuvent être effectuées une seule fois, avant de générer un ou plusieurs chemins.

Enfin, le tableau 8 montre le temps que prend la génération de 100 chemins, sans compter les étapes de prétraitement.

Table 8 : Temps mis pour générer 100 chemins sans compter l'étape de prétraitement. Cette valeur a été obtenue en mesurant le temps mis pour générer 101 chemins, et en lui retranchant le temps pour la génération d'un chemin

	200	1000	2000	3000	5000	8000
test 2	0.4s	0.9s	2.4s	4.3s	10.2s	23.5s
test 3	0.3s	0.8s	2.3s	3.3s	7.7s	12.9s
test 4	0.0s	1.1s	1.6s	2.8s	6.2s	13.8s
test 5	0.6s	0.2s	1.9s	1.8s	5.5s	12.3s
test 6	0.0s	0.6s	1.6s	1.9s	5.1s	12.0s
test 7	0.6s	0.0s	2.9s	2.7s	5.4s	10.7s
test 10	0.3s	0.3s	1.3s	2.8s	4.6s	9.7s

Ces valeurs prouvent que la génération est rapide même lorsque le modèle a plus de 10^{24} états, car seule compte la taille des modules qui le composent (ici 289 états). On remarquera aussi que le temps de génération diminue lorsque le nombre de modules qui composent le modèle augmente. Ceci s'explique par le fait que plus il y a de modules, moins la longueur des chemins à tirer dans chaque module est grande.

Finalement, ces expériences prouvent que la génération de chemins à partir des modules qui composent le modèle permet d'explorer uniformément des modèles de très grande taille.

Dans le cadre de ce stage, les expériences qui ont été menées et qui sont présentées dans les tables 6 à 8 n'ont pas révélées les limites, en termes de taille des modèles et de longueur des chemins, de notre implémentation du tirage uniforme dans des systèmes asynchrones.

5. TIRAGE APPROCHÉ AVEC SYNCHRONISATION

Dans la section précédente, tous les modules évoluaient de manière indépendante. En pratique, cette hypothèse est rarement vérifiée car les modules doivent souvent en attendre d'autres avant de pouvoir emprunter une transition ; ces transitions spéciales sont appelées des transitions synchronisées. Nous allons maintenant supposer que chaque module comporte exactement une transition synchronisée, étiquetée α. Ainsi, dans le système global, tous les modules doivent prendre la transition α en même temps. Ce cas a déjà été traité dans [5]. Cette section explique le fonctionnement de l'algorithme que j'ai implémenté, puis nous analyserons les tests de performances. Le cas plus général où plusieurs transitions synchronisées sont présentes dans chaque module sera étudié dans la section suivante. Mais pour un souci de clarté, nous préférons expliquer d'abord le fonctionnement avec une seule synchronisation.

5.1 Principe

Soit α le symbole de synchronisation, p_α (resp. s_α) l'état précédant (resp. suivant) la transition α d'un module, et w un chemin du modèle global. Si w contient m synchronisations, alors $w = w_0\ \alpha\ w_1\ \alpha \cdots \alpha\ w_m$ et le symbole α n'est présent dans aucun des w_i. Ainsi, on peut utiliser l'algorithme de la section 4 pour tirer les $m+1$ sous chemins w_i, à condition de les tirer dans des sous langages des modules M_i. En effet, w_0 doit commencer aux états initiaux de chacun des modules, finir aux p_α, et bien sûr ne jamais emprunter une transition α. w_1 lui par contre doit commencer aux s_α et finir aussi aux p_α. Au final, il faut considérer 4 langages par module :

T_i **:** l'état initial et les états finaux sont ceux de M_i,

B_i **:** l'état initial est celui de M_i, et l'état final est p_α,

E_i **:** l'état initial est s_α, et les états finaux sont ceux de M_i,

C_i **:** l'état initial est s_α, et l'état final est p_α.

B_i, C_i, E_i et T_i ne contiennent aucune transition synchronisée.

Donc chaque module doit être décomposé en 4 sous modules. La figure 4 montre un exemple de module avec ses 4 décompositions.

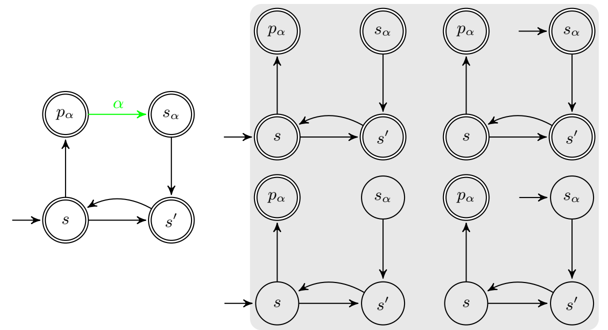

Figure 4 : Un module M_i qui contient une transition synchronisé, étiquetée α, et ses quatre modules asynchrones associés.

Finalement, si on calcule $l(n)$ (resp. $l(n,m)$), le nombre de chemins de longueur n (resp. qui contiennent m synchronisations), alors on obtient l'algorithme suivant de génération d'un chemin de longueur n :

1. choisir m avec probabilité :

$$P(m) = \frac{l(n,m)}{l(n)},$$

2. choisir les longueurs $(i_0, \dots, i_m)$ des chemins $w_0, \dots, w_m$ avec une probabilité appropriée,
3. générer chaque chemin w_k de longueur i_k dans les sous modules correspondants en utilisant l'algorithme de génération uniforme de chemins dans des systèmes asynchrones qui a été vu à la section 4.

Le chemin final est $w = w_0\ \alpha\ w_1\ \alpha \dots \alpha\ w_m$.

Il reste encore à trouver une méthode pour calculer efficacement les $l(n)$ et $l(n,m)$. En sachant que :

$$l(n) = \sum_{m=0}^{n} l(n,m),$$

il suffit de savoir calculer $l(n,m)$ pour tout $0 \leq m \leq n$. Ensuite nous verrons quelle probabilité utiliser pour choisir convenablement les longueurs $(i_0, \ldots, i_m)$.

5.1.1 *Calcul de* $l(n,m)$

Soit $b(n)$ (resp. $b_i(n)$, $c(n)$, $c_i(n)$, $e(n)$, $e_i(n)$, $t(n)$, $t_i(n)$) le nombre de chemins de longueur n dans B (resp. B_i, C, C_i, E, E_i, T, T_i). Alors si $l(n,m,i_0,i_m)$ désigne le nombre de chemins de longueur n contenant m synchronisations et avec w_0 (resp. w_m) qui est de longueur i_0 (resp. i_m), on a :

$$l(n,m) = \begin{cases} t(n) & \text{si } m = 0, \\ \displaystyle\sum_{i_0+i_m=n-m} l(n,m,i_0,i_m) & \text{sinon.} \end{cases}$$

et pour tout $m > 0$:

$$l(n,m,i_0,i_m) = b(i_0)e(i_m) \sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} c(i_1)\ldots c(i_{m-1})$$

Comme lorsqu'on cherchait à calculer $l(n)$ à la section 4, les valeurs $b(n)$, $c(n)$, $e(n)$ et $t(n)$ font intervenir le modèle global mais elles peuvent être estimées à partir des constantes ω_i et C_i calculées dans les différents sous modules. Ainsi, en considérant :

$$\begin{array}{rcl} t_i(n) & \sim & C_{t,i}\ (\omega_{t,i})^n, \\ b_i(n) & \sim & C_{b,i}\ (\omega_{b,i})^n, \\ e_i(n) & \sim & C_{e,i}\ (\omega_{e,i})^n, \\ c_i(n) & \sim & C_{c,i}\ (\omega_{c,i})^n. \end{array}$$

On obtient les approximations suivantes :

$$t(n) \sim C_{t,1}\ldots C_{t,r}(\omega_{t,1}+\cdots+\omega_{t,r})^n, \quad (1a)$$
$$b(n) \sim C_{b,1}\ldots C_{b,r}(\omega_{b,1}+\cdots+\omega_{b,r})^n, \quad (1b)$$
$$e(n) \sim C_{e,1}\ldots C_{e,r}(\omega_{e,1}+\cdots+\omega_{e,r})^n, \quad (1c)$$
$$c(n) \sim C_{c,1}\ldots C_{c,r}(\omega_{c,1}+\cdots+\omega_{c,r})^n. \quad (1d)$$

Par conséquent, pour tout $m > 0$:

$$l(n,m,i_0,i_m) \sim (C_{b,1}\ldots C_{b,r})(C_{c,1}\ldots C_{c,r})^{m-1}(C_{e,1}\ldots C_{e,r}) (\omega_{b,1}+\cdots+\omega_{b,r})^{i_0} (\omega_{c,1}+\cdots+\omega_{c,r})^{n-m-i_0-i_m} (\omega_{e,1}+\cdots+\omega_{e,r})^{i_m} \binom{n-2-i_0-i_m}{m-2}. \quad (2)$$

Le calcul de $l(n,m,i_0,i_m)$ nécessite $O(n(m+r))$ opérations arithmétiques.

La preuve de la formule (2) est donnée à l'appendice A.

Calculer tous les $l(n,m,i_0,i_m)$ pour toutes les valeurs de m telles que $0 \leq m \leq n$ et toutes les paires (i_0,i_m) telles que $0 \leq i_0+i_m \leq n-m$ demande $O(n^3 \times rn + n^2 \times n^2) = O(rn^4)$ opérations arithmétiques (à condition de garder en mémoire une table de tous les coefficients binomiaux $\binom{k}{p}$ pour $0 \leq k \leq n$ et $0 \leq p \leq k$, afin d'éviter de recalculer plusieurs fois le même coefficient)

5.1.2 *Choix des longueurs* $(i_0, \ldots, i_m)$

Une fois que le nombre de synchronisations - m - est décidé et dans le cas où il est supérieur à 0 (sinon on retombe sur le cas asynchrone et il suffit d'appliquer l'algorithme de la section 4), il reste à déterminer les longueurs $(i_0, \ldots, i_m)$ des chemins à tirer $(w_0, \ldots, w_m)$ telles que $i_0+\cdots+i_m = n-m$.

On commence par choisir i_0 et i_m avec la probabilité :

$$P(i_0,i_m) = \frac{l(n,m,i_0,i_m)}{l(n,m)}$$

Ce qui nécessite au pire $O(n^2)$ opérations arithmétiques.

Puis on tire un $(m-1)$-uplet $(i_1, \ldots, i_{m-1})$ avec probabilité :

$$P(i_1,\ldots,i_{m-1}) = \frac{c(i_1)\ldots c(i_{m-1})}{\sum_K c(k_1)\ldots c(k_{m-1})}$$

avec K qui représente l'ensemble des k_i tels que :

$$k_1 + \cdots + k_{m-1} = n - m - i_0 - i_m$$

En se servant de la formule (1d), cette probabilité se réduit en :

$$P(i_1,\ldots,i_{m-1}) \sim \frac{1}{\binom{n-2-i_0-i_m}{m-2}} \quad (3)$$

La preuve de la formule (3) est donné à l'appendice B. Un algorithme assez simple permet de choisir $(i_1, \ldots, i_m)$ avec cette probabilité :

1. choisir $m-2$ nombres distincts $(j_1, \ldots, j_{m-2})$ tous compris entre 0 et $n-2-i_0-i_m$ inclus,
2. les ranger par ordre croissant : $j_1 < j_2 < \cdots < j_{m-2}$,
3. $i_1 = j_1$, $i_2 = j_2 - j_1$, …, $i_{m-2} = j_{m-2} - j_{m-1}$ et $i_{m-1} = n-2-i_0-i_m-j_{m-2}$.

La complexité de cet algorithme est en $O(n \times m)$.

Au final, le pré-traitement pour choisir m demande $O(rn^4)$ opérations arithmétiques afin de calculer tous les $l(n,m)$ et $l(n)$. Ensuite, le choix de m nécessite au pire $O(n)$ opérations arithmétiques. Puis le choix des longueurs requiert au pire $O(n^2)$ opérations arithmétiques.

La génération uniforme de chemins dans un système concurrent avec une seule transition synchronisée par module est quadratique en la longueur n du chemin à générer et elle est linéaire (en temps et en mémoire) en la taille du plus grand module.

5.2 Mesures et résultats

Les modèles des systèmes asynchrones (cf. table 6) ont été réutilisés ici, en ré-étiquetant une transition dans chaque module par l'étiquette de synchronisation α.

La table 9 montre les temps de prétraitement et la table 10 nous dévoile le temps nécessaire pour générer 100 chemins. Le symbole ∞ signifie qu'il n'y avait pas assez de mémoire pour calculer tous les $l(n, m)$ et $l(n)$.

Table 9 : Temps mis pour l'étape de *prétraitement*

	100	200	300	400	500	600
test 2	3m10	3m25	4m12	5m55	9m06	∞
test 3	4m45	5m02	5m54	7m46	11m11	∞
test 4	6m19	6m39	7m34	9m34	13m13	∞
test 5	7m51	8m16	9m15	11m23	15m18	∞
test 6	9m24	9m50	10m56	13m12	17m21	∞
test 7	11m00	11m27	12m35	15m03	19m24	∞
test 10	12m28	12m31	12m33	12m32	20m26	∞

Table 10 : Temps mis pour générer 100 chemins, sans compter la phase de *prétraitement*

	100	200	300	400	500	600
test 2	1.9s	2.1s	4.3s	4.7s	4.4s	∞
test 3	0.9s	1.2s	1.2s	2.6s	3.8s	∞
test 4	1.5s	1.5s	0.9s	3.2s	2.5s	∞
test 5	3.0s	0.4s	0.9s	3.1s	2.5s	∞
test 6	4.1s	0.7s	0.9s	3.8s	2.5s	∞
test 7	3.7s	2.9s	1.9s	1.6s	2.4s	∞
test 10	4.6s	0.6s	0.8s	1.5s	2.2s	∞

Ces expériences démontrent la faisabilité du tirage de chemins dans des très grands modèles, et cela même en présence de synchronisation entre les modules concurrents qui décrivent le modèle global. Le cas où il y a plusieurs synchronisations étiquetées par différents symboles est plus complexe mais un algorithme est expliqué à la section suivante (à défaut d'avoir des tests de performance qui sont en cours de réalisation).

6. SYNCHRONISATIONS MULTIPLES

Les sections précédentes expliquaient les résultats de la partie pratique de mon stage qui consistait à implémenter les solutions proposées dans [5]. Dans cette section, nous généralisons l'algorithme de la section 5 en autorisant plusieurs synchronisations étiquetées par différents symboles. La seule hypothèse restante est que tous les modules possèdent au moins une occurrence de chaque étiquette de synchronisation. Sinon nous serions en présence de *synchronisation partielle* ce qui, comme nous le verrons à la section 7, nous compliquerait encore la tâche. On commencera par expliquer le cas où chaque module contient plusieurs transitions étiquetées par le même symbole de synchronisation, avant de traiter le cas avec différentes étiquettes de synchronisation.

6.1 Chaque module contient plusieurs transitions synchronisées

Soit α l'étiquette de synchronisation. Sans perte de généralité, nous supposerons que chaque module possède k transitions étiquetées α (dans le but de simplifier les notations et les calculs de complexité). Par exemple, la figure 5 montre un module avec deux transitions étiquetées par α. Chaque état précédant une transition α est nommé p_{α_i} avec un indice i pour distinguer les occurrences de transitions α. De même, s_{α_i} désigne l'état suivant une transition α. Les différentes occurrences de l'étiquette α dans un module ne sont pas différenciées dans une trace, cependant il est nécessaire de les distinguer pour savoir dans quels sous modules doivent être tirés les chemins qui précèdent et suivent une étiquette α. En effet, si un chemin précédant une étiquette α se termine sur l'état p_{α_1} du module représenté sur la figure 5, alors il faut que le chemin qui suit cette étiquette α commence, dans ce module, par l'état s_{α_1}.

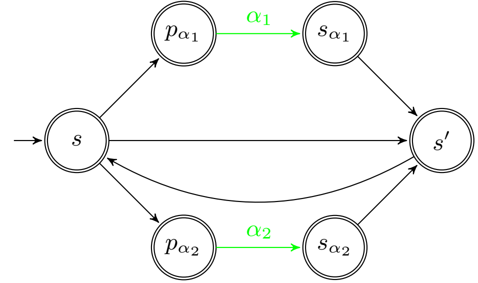

Figure 5 : un module M_i contenant deux transitions synchronisées α, qui sont ici distinguées en α_1 et α_2.

Plaçons nous à la i-ème synchronisation d'un chemin w : $w = \dots w_{i-1}\ \alpha\ w_i \dots$. Cette étiquette α représente en réalité r étiquettes α (une par module). Or il existe k transitions α par module. Donc il y a k^r possibilités pour cette i-ème synchronisation. C'est l'unique différence avec le problème de la section 5. Nous allons maintenant décrire l'algorithme qui prend en compte ces k transitions synchronisées par module.

Soit G_i le graphe sous-jacent au module M_i, on note G_i^α le graphe G_i sans les transitions étiquetées α.

Les langages suivants sont construits à partir de G_i^α, et se distinguent uniquement par leurs états initiaux et finaux :

T^i : l'état initial et les états finaux sont ceux de M_i,

B_j^i : l'état initial est celui de M_i, et l'état final est p_{α_j} $(\forall j \in [1, k])$,

E_j^i : l'état initial est s_{α_j}, et les états finaux sont ceux de M_i $(\forall j \in [1, k])$,

$L_{j,j'}^i$: l'état initial est s_{α_j}, et l'état final est $p_{\alpha_{j'}}$ $(\forall j, j' \in [1, k])$

La figure 6 donne un exemple de sous module représentant l'un de ces langages.

On a, par module, $(k + 1)^2$ langages différents à considérer. Cependant, seules $k + 1$ tables de comptage (du nombre de chemins de longueur n) doivent être construites, car une table calcule tous les chemins de longueur inférieure ou égale à n pour tous les sommets. Le langage E_1^i aura ainsi la même table de comptage que le langage E_2^i. Malgré tout, dans l'état actuel du programme C, généré par MuPAD pour effectuer le tirage de chemins, on ne peut pas choisir le sommet de départ. La fonction dans MuPAD qui crée ce programme

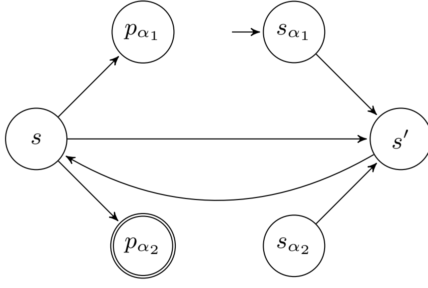

Figure 6 : Le sous module correspondant au langage $L^i_{1,2}$ du module décrit par la figure 5

C est en cours de réécriture afin de se passer de MuPAD, reconstruire les tables par une formule triviale de récurrence (cf. formule (3.1)), et tirer des chemins en faisant une marche aléatoire de longueur n guidée par ces tables).

Maintenant, posons $A_m \in [1..k]^{r\times m}$ la matrice définie telle que $a_{i,j} = A_m(i,j)$ correspond à l'occurrence de transition α dans le module M_i qui est utilisée à la j-ème synchronisation d'un chemin contenant m synchronisations. Cette matrice permet de définir précisément quelles transitions synchronisées sont utilisées dans un chemin comportant m occurrences d'α.

Par exemple, supposons que nous ayons un système composé de deux modules concurrents $M1$ et $M2$ qui sont tous les deux identiques au module décrit à la figure 5. Soit w un chemin de longueur 10 comportant 3 synchronisations : $w = w_0\ \alpha\ w_1\ \alpha\ w_2\ \alpha\ w_3$. Une des 64 matrices A_3 possibles est :

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \end{pmatrix}$$

Ce qui signifie que : w_0 est le mélange d'un chemin de $M1$ allant de s à p_{α_2} et d'un chemin de $M2$ allant de s à p_{α_1}. w_1 est le mélange d'un chemin de $M1$ allant de s_{α_2} à p_{α_1} et d'un chemin de $M2$ allant de s_{α_1} à p_{α_2}. w_2 est le mélange d'un chemin de $M1$ allant de s_{α_1} à p_{α_1} et d'un chemin de $M2$ allant de s_{α_2} à p_{α_1}. Et enfin w_3 est le mélange d'un chemin allant de s_{α_1} à n'importe quel état de $M1$ et d'un chemin allant de s_{α_1} à n'importe quel état de $M2$.

Si on calcule $l(n)$ (resp. $l(n, A_m)$), le nombre de chemins de longueur n (resp. qui passent par les m synchronisations définies par A_m), alors on retombe sur un algorithme, quasiment identique à celui défini à la section 5, qui permet la génération d'un chemin de longueur n : la seule différence étant de choisir une matrice A_m au lieu d'un nombre m, avec la probabilité :

$$P(A_m) = \frac{l(n, A_m)}{l(n)}$$

Il reste à trouver une manière efficace de calculer les $l(n, A_m)$ et $l(n)$.

6.2 Calcul des $l(n, A_m)$

Soit w un chemin de longueur n qui passe par les synchronisations définies par A_m, alors $w = w_0.\vec{\alpha_1}.w_1 \dots \vec{\alpha_m}.w_m$, où $\vec{\alpha_j}$ est la j-ème colonne de A_m. De plus, en considérant $l^i(n)$ (resp. $t(n)$, $t^i(n)$, $b_j(n)$, $b^i_j(n)$, $e_j(n)$, $e^i_j(n)$, $l_{j,j'}(n)$, $l^i_{j,j'}(n)$) le nombre de mots de longueur n dans L^i (resp. T, T^i, $B_{a_{.,j}}$, $B^i_{a_{i,j}}$, $E_{a_{.,j}}$, $E^i_{a_{i,j}}$, $L_{a_{.,j},a_{.,j'}}$, $L^i_{a_{i,j},a_{i,j'}}$). On a alors :

$$l(n) = \sum_{m=0}^{n} \sum_{A_m \in [1..k]^{r\times m}} l(n, A_m)$$

où

$$l(n, A_m) = \begin{cases} t(n) & \text{si } m = 0, \\ \sum_{i_0+\dots+i_m = n-m} l(n, A_m, I) & \text{sinon.} \end{cases}$$

et pour $m > 0$ et $I = (i_0, \dots, i_m)$:

$$l(n, A_m, I) = b_1(i_0).l_{1,2}(i_1) \dots l_{m-1,m}(i_{m-1}).e_m(i_m)$$

En supposant que tous les T^i, les B^i_j, les E^i_j, et les $L^i_{j,j'}$ respectent les conditions d'apériodicité et de forte connexité [6], on a alors :

$$\begin{aligned} t^i(n) &\sim C^i_t\ (\omega^i_t)^n, \\ b^i_j(n) &\sim C^i_{b,j}\ (\omega^i_{b,j})^n, \\ e^i_j(n) &\sim C^i_{j,e}\ (\omega^i_{j,e})^n, \\ l^i_{j,j'}(n) &\sim C^i_{j,j'}\ (\omega^i_{j,j'})^n. \end{aligned}$$

où $C^i_{j,j'}$ et $\omega^i_{j,j'}$ désignent respectivement la constante C et la valeur ω calculées dans le sous module $L^i_{a_{i,j},a_{i,j'}}$.

Les valeurs qui font intervenir le modèle global peuvent être approximées par :

$$\begin{aligned} t(n) &\sim C^1_t \dots C^r_t (\omega^1_t + \dots + \omega^r_t)^n, \\ b_j(n) &\sim C^1_{b,j} \dots C^r_{b,j} (\omega^1_{b,j} + \dots + \omega^r_{b,j})^n, \\ e_j(n) &\sim C^1_{j,e} \dots C^r_{j,e} (\omega^1_{j,e} + \dots + \omega^r_{j,e})^n, \\ l_{J,J'}(n) &\sim C^1_{j,j'} \dots C^r_{j,j'} (\omega^1_{j,j'} + \dots + \omega^r_{j,j'})^n. \end{aligned}$$

On peut ainsi approximer $l(n, A_m)$ à partir uniquement des r modules :

$$\begin{aligned} l(n, A_m) \sim\ & (C^1_{b,1} \dots C^r_{b,1}) \prod_{v=1}^{m-1} (C^1_{v,v+1} \dots C^r_{v,v+1}) \\ & (C^1_{m,e} \dots C^r_{m,e}) \\ & \sum_{i_0+\dots+i_m = n-m} \Big\{ (\omega^1_{b,1} + \dots + \omega^r_{b,1})^{i_0} \\ & \prod_{v=1}^{m-1} (\omega^1_{v,v+1} + \dots + \omega^r_{v,v+1})^{i_v} \\ & (\omega^1_{m,e} + \dots + \omega^r_{m,e})^{i_m} \Big\} \end{aligned} \quad (4)$$

Le calcul de $l(n, A_m)$ nécessite $O(n^2 r)$ opérations arithmétiques.

La preuve de la formule (4), ainsi que le calcul du nombre d'opérations nécessaires pour calculer tous les $l(n, A_m)$ et $l(n)$ est disponible en appendice C.

6.3 L'algorithme de génération uniforme

Maintenant, on peut définir un algorithme pour la génération de chemins de longueur n.

1. En utilisant la formule (4), calculer $l(n, A_m)$ pour toutes les valeurs possibles de A_m. Ce qui requiert, au pire, $O(n^2 rk^{r\times n})$ opérations arithmétiques.
2. Choisir une séquence A_m de transitions α avec probabilité :
$$Pr(A_m) = \frac{l(n, A_m)}{l(n)}$$
Calculer ces probabilités demande $O(k^{r\times n})$ opérations arithmétiques.
3. Si $m = 0$, alors générer uniformément un chemin de longueur n dans le langage T en utilisant le même algorithme que dans la section 4.
4. Si $m > 0$, alors choisir les longueurs des mots $w_0, \ldots, w_m$ en prenant un $(m+1)$-uplet $I = (i_0, \ldots, i_m)$ avec probabilité :
$$Pr(I) = \frac{l(n, A_m, I)}{l(n, A_m)}$$
Puis générer les w_j de longueur i_j en effectuant un mélange des r chemins tirés dans les r langages appropriés.

Au final, le pré-traitement pour choisir A_m demande $O(n^2 rk^{r\times n})$ opérations arithmétiques afin de calculer tous les $l(n, A_m)$ et $l(n)$. Ensuite, le choix de A_m nécessite au pire $O(k^{r\times n})$ opérations arithmétiques. Puis le choix des longueurs requiert au pire $O(n^2)$ opérations arithmétiques.

6.4 Plusieurs étiquettes de synchronisation par module

Avec une seule étiquette de synchronisation, nous avons déjà distingué les différentes transitions synchronisées. Il est possible d'utiliser le même algorithme pour le cas plus général où il existe plusieurs étiquettes de synchronisation et ceci en ne modifiant pratiquement rien. Prenons l'exemple d'un système constitué de plusieurs exemplaires du module décrit à la figure 7 ; ce système contient 2 étiquettes de synchronisation : α et β. Soit w un chemin dans ce système comportant m synchronisations que nous noterons par un *méta-symbole* σ : $w = w_0\ \sigma\ w_1\ \sigma \ldots \sigma\ w_m$. Chaque symbole σ peut représenter n'importe quel symbole de synchronisation (ici, α ou β). Il suffit de tester toutes les combinaisons possibles en prenant soin à chaque fois de choisir dans chaque module une transition parmi celles étiquetées par le bon symbole de synchronisation. Dans le cas de la figure 7, l'ensemble des matrices A_m à considérer est inclus strictement dans $[1..5]^{r\times m}$, car il est impossible de faire une transition α dans un module en même temps qu'un autre module exécute une transition β.

La construction des matrices A_m est majorée par $O(n^2 rk^{r\times n})$ opérations arithmétiques, où k désigne le nombre de transitions synchronisées sans distinction des étiquettes de synchronisation.

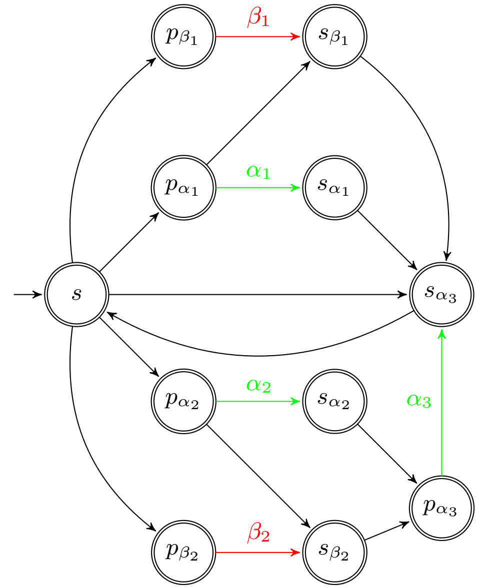

Figure 7 : Un module comportant 5 transitions synchronisées étiquetées par 2 symboles de synchronisation : α et β.

6.5 Conclusion pour les synchronisations multiples

On vient de voir que la génération uniforme de chemins dans un système concurrent avec plusieurs transitions synchronisées par module est exponentielle en la longueur n du chemin à générer. Mais l'exponentielle dépend uniquement du nombre de transitions synchronisées alors que dans l'algorithme de la section 3 on était exponentielle en la taille du plus grand module. Le gain est relatif au ratio entre le nombre de transitions synchronisées et le nombre total de transitions : plus ce ratio est petit, plus cet algorithme est performant par rapport à tirer uniformément à partir du modèle global. Ce qui signifie que si le nombre de transitions synchronisées présentes dans un module est borné par une valeur $k << |M|$, alors cet algorithme rend possible l'exploration de très grands modèles qui sont décrits par des modules concurrents avec peu de synchronisations, communes à tous les modules. Les tests de performance de cet algorithme sont en cours de réalisation.

7. PROBLÈME DE LA SYNCHRONISATION PARTIELLE

La section précédente a montré la faisabilité de la génération uniforme de chemins à partir des modules concurrents décrivant un système global dans le cas où les modules se donnent tous *rendez-vous* à certains moments. Il reste un cas qui n'a pas été traité et qui est celui de la *synchronisation partielle*.

Le problème de la synchronisation partielle (par opposition à la synchronisation commune à tous les modules) survient dès qu'au moins un module ne contient pas de transition étiquetée par l'un des symboles de synchronisation. Par

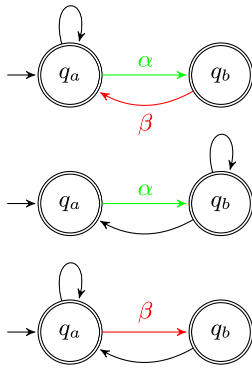

Figure 8 : Trois modules concurrents M_1, M_2 et M_3 comportant des transitions synchronisées α et/ou β.

exemple le module M_2 (cf. figure 8) n'a aucune transition étiquetée β. Dans ce cas, la décomposition d'un chemin de longueur n contenant m synchronisations en $m+1$ chemins est problématique. Prenons par exemple un chemin contenant une synchronisation α suivi d'une synchronisation β : $w = w_0.\alpha.w_1.\beta.w_2$. Le sous chemin w_0 est le résultat d'un mélange des chemins w_0^1, w_0^2, w_0^3 tirés respectivement dans M_1, M_2 et M_3. w_0^1 commence à l'état initial de M_1 et finit sur un état précédant une transition étiquetée α ; w_0^2 commence à l'état initial de M_2 et finit sur un état précédant une transition α. Mais w_0^3 commence à l'état initial de M_3 et peut finir n'importe où (car M_3 ne contient aucune transition α). C'est à dire que w_1^3 devra commencer depuis l'état où s'est arrêté w_0^3, qui peut être n'importe quel état de M_3.

7.1 Solution naïve pour le cas général

Soit M_i un module ne contenant pas l'une des étiquettes de synchronisation parmi l'ensemble des étiquettes de ses transitions. Soit α cette étiquette. Si on cherche à construire un chemin de longueur n contenant à un moment le symbole α : $w = w_0.\beta \dots w_j.\alpha.w_{j+1}\gamma \dots$. Pour le mot w_j, il faut considérer tous les chemins finissant sur n'importe quel état de M_i. Si M_i contient $|M_i|$ états, il faut donc considérer $|M_i|^2$ langages différents pour M_i, car chaque état peut être origine ou extrémité d'un sous chemin de w.

Dès qu'un module ne se synchronise pas sur une étiquette de synchronisation, chacun de ses états peut devenir un état final ou initial ; on utilise le même algorithme que pour le cas général de synchronisation communes à tous les modules. Bien entendu, on se retrouve avec une explosion combinatoire du nombre de langages à considérer.

7.2 Solution élégante pour une seule étiquette de synchronisation

Si il n'y a qu'une seule étiquette de synchronisation, alors on peut gérer la synchronisation partielle plus efficacement qu'avec la solution naïve de la sous-section précédente. On commence par tirer un chemin synchronisé w' avec tous les modules qui possèdent au moins une occurrence de transition synchronisée en utilisant l'algorithme de la section 6.3. Puis on tire un chemin w'' dans les modules qui n'ont aucune transition synchronisée en se servant de l'algorithme de la section 4. Enfin, w est le résultat du mélange de w' avec w'' en utilisant l'algorithme décrit à la section 4.1.4. Au final, w aura bien été tiré uniformément dans un système contenant à la fois des modules synchronisés et des modules asynchrones.

8. CONCLUSION ET PERSPECTIVES

Ce rapport dévoile les premiers résultats expérimentaux sur les marches aléatoires globalement uniformes dans de très grands systèmes qui sont décrits par un ensemble de composants concurrents et plus petits. On parle de marche aléatoire globalement uniforme lorsqu'à chaque étape de la marche, le choix du successeur est biaisé de manière à ce que tous les chemins, d'une certaine longueur dans le modèle global, aient la même probabilité d'apparaître.

La section 3 montre que la méthode naïve (qui consiste à compter le nombre de chemins de longueur choisie qui partent de chaque successeur, et à ajuster en conséquence les probabilités de choisir ces successeurs) est réalisable uniquement pour des modèle de taille moyenne : jusqu'à environ 10^4 états selon nos expériences.

Pour des systèmes plus grands, les expériences des sections 4 et 5 démontrent qu'en effectuant des tirages uniformes locaux à chaque module et en estimant le nombre de chemins du modèle global, il est possible d'explorer uniformément des modèles très grands, qui sont décrits par un ensemble de modèles concurrents et plus petits : la barre symbolique des 10^{20} états [13] a été largement franchie puisque des chemins ont été tirés sur un modèle ayant 4.10^{24} états. la complexité reste limitée à la taille du plus grand des composants. Ces résultats ont été publiés dans un *workshop* international [17].

La section 6 décrit un nouvel algorithme pour généraliser le tirage uniforme en présence de synchronisation au cas où plusieurs transitions étiquetées par différents symboles de synchronisations sont présentes. Son implémentation est en cours de réalisation.

Enfin, la section 7 aborde le problème de la synchronisation partielle qui reste un problème ouvert actuellement.

Cette méthode peut être utile pour le test aléatoire (structurel ou fonctionnel), le model-checking, ou encore la simulation de protocoles lorsqu'ils font intervenir de nombreuses entités distribuées, ce qui est souvent le cas en pratique.

9. REMERCIEMENTS
Je remercie mes encadrants Alain Denise et Marie-Claude Gaudel pour tout le temps qu'ils m'ont consacré ; toute l'équipe du groupe Génie Logiciel pour leur accueil chaleureux qui a facilité mon intégration ; sans oublier les membres du groupe RaSTa qui m'ont souvent conseillé lors de nos nombreuses réunions. Enfin, je remercie ma fiancée pour son incommensurable mansuétude.

10. REFERENCES

[1] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. MASSON, 1992.

[2] E. Brinksma and J. Tretmans. Testing transition systems : an annotated bibliography. In LNCS, editor, *MOVEP 2000*, volume 2067, pages 187–195. Springer-Verlag, 2001.

[3] C. Creutzig and W. Oevel. *MuPAD Tutorial*. Springer, second edition, 2004.

[4] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE '04 : Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 25–34, Washington, DC, USA, 2004. IEEE Computer Society.

[5] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, and S. Peyronnet. Uniform random sampling of traces in very large models. In *RT '06 : Proceedings of the 1st international workshop on Random testing*, pages 10–19, New York, NY, USA, 2006. ACM Press.

[6] P. Flajolet and R. Sedgewick. Analytic combinatorics : Functional equations, rational and algebraic functions. Research Report 4103, INRIA, 2001. 98 pages.

[7] H. Garavel and N. Descoubes. Very large transition systems. http ://tinyurl.com/yuroxx.

[8] H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001. Technical Report 0254, INRIA, 2001.

[9] H. Garavel and R. Ruffiot. Binary Coded Graphs. http ://www.inrialpes.fr/vasy/cadp/man/bcg.html.

[10] S.-D. Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique*. PhD thesis, Université Paris-Sud 11, LRI, june 2004.

[11] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *ASE '01 : Proceedings of the 16th IEEE international conference on Automated software engineering*, page 5, Washington, DC, USA, 2001. IEEE Computer Society.

[12] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics : A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[14] D. Lee and M. Yannakakis. Principles and methods of testing finite state machine - a survey. In *The Proceedings of IEEE*, volume 84, pages 1089–1123, August 1996.

[15] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill Book Company, 1996.

[16] J. Maurer, D. Abrahams, B. Dawes, and R. Rivera. Boost random number library. http ://www.boost.org/libs/random/.

[17] J. Oudinet. Uniform random walks in very large models. In *RT '07 : Proceedings of the 2nd international workshop on Random testing*, Atlanta, GA, USA, 2007. ACM Press.

[18] N. M. Thiéry. Mupad-combinat algebraic combinatorics package for mupad. http ://mupad-combinat.sourceforge.net/.

[19] W. T. Vetterling and B. P. Flannery. *Numerical Recipes in C++ : The Art of Scientific Computing*. Cambridge University Press, second edition, 2002.

APPENDICES

A. APPROXIMATION DE $L(N, M, I_0, I_M)$

Cette appendice prouve comment obtenir l'approximation de $l(n, m, i_0, i_m)$ donnée à la formule (2). Nous rappelons que $l(n, m, i_0, i_m)$ représente le nombre de chemins de longueur n qui contiennent m synchronisation et dont le sous chemin w_0 qui va de l'état initial à l'état précédant la transition synchronisé est de longueur i_0 et le sous chemin w_m est de longueur i_m. Ainsi, pour tout $m > 0$, la valeur exacte de $l(n, m, i_0, i_m)$ est :

$$l(n, m, i_0, i_m) = b(i_0)e(i_m) \sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} c(i_1)\ldots c(i_{m-1})$$

Et en se servant des approximations (1), on obtient :

$$\begin{aligned} l(n, m, i_0, i_m) &\sim b(i_0)e(i_m) \\ &\sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} (C_{c,1}\ldots C_{c,r})^{m-1}(\omega_{c,1}+\cdots+\omega_{c,r})^{i_1+\cdots+i_{m-1}} \\ &\sim b(i_0)e(i_m)(C_{c,1}\ldots C_{c,r})^{m-1} \\ &\sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} (\omega_{c,1}+\cdots+\omega_{c,r})^{n-m-i_0-i_m} \\ &\sim b(i_0)e(i_m)(C_{c,1}\ldots C_{c,r})^{m-1} \\ &(\omega_{c,1}+\cdots+\omega_{c,r})^{n-m-i_0-i_m} \sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} 1 \end{aligned} \quad (5)$$

Or, compter le nombre de manière d'avoir $i_1 + \cdots + i_k = n$ peut se faire très facilement si on utilise la représentation unaire pour tous les i_j. Dans ce cas, on a $k - 1$ symboles '+' et n symboles '—' et donc le nombre de combinaisons possibles est : $\binom{n+k-1}{k-1}$.

On a alors :

$$\sum_{\substack{i_1+\cdots+i_{m-1}= \\ n-m-i_0-i_m}} 1 = \binom{n-2-i_0-i_m}{m-2}$$

Ce qui nous donne bien la formule (2).

B. APPROXIMATION DE $P(I_1, \dots, I_{M-1})$

$(i_1, \dots, i_{m-1})$ correspond aux longueurs respectives des chemins $w_1, \dots, w_{m-1}$ qui sont le résultat des mélanges de chemins tirés dans le langage centrale de chaque module. la probabilité exacte de ce $(m-1)$-uplet est :

$$P(i_1, \dots, i_{m-1}) = \frac{c(i_1) \dots c(i_{m-1})}{\sum_P c(k_1) \dots c(k_{m-1})}$$

avec P qui représente :

$$k_1 + \dots + k_{m-1} = n - m - i_0 - i_m$$

En utilisant la formule (3), on arrive à :

$$\begin{aligned} P(i_1, \dots, i_{m-1}) &\sim \frac{(\omega_{c,1} + \dots + \omega_{c,r})^{n-m-i_0-i_m}}{(\sum_P \omega_{c,1} + \dots + \omega_{c,r})^{n-m-i_0-i_m}} \\ &= \frac{1}{\sum_P 1}. \end{aligned}$$

Comme expliqué à l'appendice A, le nombre de possibilités de choisir k nombres tels que leur somme est égale à n est : $\binom{n+k-1}{k-1}$. Ainsi on obtient bien la formule (3) :

$$P(i_1, \dots, i_{m-1}) \sim \frac{1}{\binom{n-2-i_0-i_m}{m-2}}$$

C. APPROXIMATION DE $L(N, A_M)$

La difficulté de la formule (4) est le produit de termes dans la somme qui ne peut se simplifier contrairement au cas avec une seule synchronisation. Nous avons donc besoin de trouver une méthode pour calculer efficacement :

$$\sum_{i_1 + \dots + i_k = n} a_1^{i_1} a_2^{i_2} \dots a_k^{i_k}$$

Une première idée consiste à transformer cette somme en une relation de récurrence tel que le n-ème terme soit le résultat de cette somme. Soit $P_k(n)$ cette relation de récurrence :

$$\begin{cases} P_k(0) &= 1 \\ P_k(n) &= \frac{\left(\sum_{j=1}^{k} a_j\right) P_k(n-1) \; + \; \sum_{j=1}^{k} a_j^n}{2} \end{cases}$$

On a bien :

$$P_k(n) = \sum_{i_1 + \dots + i_k = n} a_1^{i_1} a_2^{i_2} \dots a_k^{i_k}$$

Il reste encore à trouver une formule close à cette relation de récurrence en espérant qu'elle soit suffisamment élégante pour être calculée rapidement. En utilisant une méthode dans [12, chap. 2], je suis arrivé à trouver la formule close suivante :

$$P_k(n) = \frac{(a_1 + \dots + a_k)^n}{2^{n+1}} \left(2 + \sum_{j=1}^{n} \frac{2^j (a_1^i + \dots + a_k^i)}{(a_1 + \dots + a_k)^i} \right)$$

Malheureusement, elle est aussi coûteuse que la relation de récurrence. Nous nous servirons donc de la relation de récurrence suivante pour calculer la formule (4). Ce qui nous donne une complexité arithmétique en $O(n^2)$.