Analyse quantitative de programmes applicatifs à base de squelettes algorithmiques

Anne Benoit, Murray Cole, Stephen Gilmore & Jane Hillston

School of Informatics
University of Edinburgh
Écosse
enhancers@inf.ed.ac.uk

Résumé

Nous proposons dans cet article une nouvelle technique pour analyser quantitativement des programmes structurés à l'aide de squelettes algorithmiques. Nous donnons un aperçu de la programmation parallèle structurée de haut niveau et nous motivons son utilité et sa pertinence pour les programmes parallèles applicatifs. Nous introduisons ensuite quelques concepts d'évaluation de performance, et notamment les algèbres de processus pour l'évaluation des performances PEPA (Performance Evaluation Process Algebra). Ces présentations nous permettent d'introduire les modèles en PEPA de deux squelettes classiques, les squelettes *Pipeline* et *Donne*. Enfin, nous décrivons un simulateur, écrit en Objective Caml, permettant de déboguer des modèles PEPA pas à pas, et nous illustrons l'utilisation du simulateur sur nos modèles de squelettes.

1. Introduction

Les langages de programmation applicatifs ont beaucoup de vertus. Ils combinent des systèmes robustes, descriptifs, à typage statique ; une analyse précise des fonctions lors de la compilation ; des facilités pour structurer des programmes à un haut niveau ; une sémantique soigneusement considérée pour le langage ; et une vision régulière et prévisible des fonctions et des données comme valeurs de première classe qui favorisent la réutilisation de code une fois couplées à un système de type polymorphe.

Les programmes écrits avec des langages applicatifs peuvent être concis, facilitant le processus d'audit et d'inspection du code pour détecter des erreurs de programmation. Tous ces avantages permettent aux développeurs de créer rapidement des applications de haute qualité qui accomplissent des tâches de traitement symbolique complexes. Comparés à des applications similaires mais développées dans des langages de programmation n'ayant pas ces propriétés, les programmes applicatifs s'avèrent souvent n'avoir que relativement peu d'erreurs de programmation. Dans la communauté de l'informatique théorique, de grands espoirs sont fondés sur les langages de programmation applicatifs. Ils sont souvent choisis pour implémenter des logiciels de preuve de théorèmes et autres outils de vérification pour lesquels la correction du programme est primordiale [28].

Si les bénéfices tirés de l'utilisation des langages de programmation applicatifs sont incontestables, ces derniers présentent cependant un désavantage. Il est en effet autant, voire plus difficile d'implémenter un support d'exécution pour le style de programmation fonctionnelle pure que pour un modèle de programmation impérative. Pour des raisons de pragmatisme [28], les langages fonctionnels comme ceux de la famille ML incluent des structures de données issues des langages impératifs comme les tableaux et les références, ainsi que des constructeurs de boucle pour permettre d'exprimer des répétitions itérativement et non récursivement. Ces propriétés ont l'avantage de permettre l'implémentation directe d'algorithmes impératifs et de sauver de la place mémoire en remplaçant des fonctions récursives par des itérations s'exécutant avec une taille de pile constante. Les programmeurs souhaitant obtenir une efficacité maximale sont en effet encouragés à utiliser le style de programmation impératif plutôt que d'abuser des fonctions récursives [26].

Le modèle de programmation impératif peut également être inadéquat cependant, pour les tâches les plus coûteuses. Pour de tels problèmes, les développeurs se tournent alors vers les programmes à exécution parallèle. Malheureusement, l'introduction arbitraire de parallélisme dans les langages de programmation applicatifs peut augmenter les chances d'avoir un comportement aléatoire non désiré des programmes, perdant ainsi les propriétés de fiabilité propres aux langages applicatifs. Lors de la conception de langages applicatifs parallèles, une attention particulière a donc été portée sur la programmation parallèle *structurée* [29], basée sur l'identification de *squelettes algorithmiques* [8]. Ces squelettes sont l'analogue en programmation parallèle structurée des fonctions polymorphes d'ordre supérieur (*map*, *fold*, et autres) qui forment la base de la programmation fonctionnelle séquentielle. Une caractéristique des squelettes est qu'ils peuvent être composés arbitrairement, aboutissant à un modèle de programmation naturel qui rappelle la nature régulière et prévisible de la composition de fonctions en programmation fonctionnelle séquentielle.

Les programmes parallèles fonctionnels structurés ont des intérêts *qualitatifs*. Par exemple, il est certain que de tels programmes n'ont pas d'interblocages [16, 27], et il est même possible de faire des preuves de programmes pour certains langages parallèles [17]. En revanche, l'utilisation d'une approche de programmation parallèle structurée ne produit pas de garanties *quantitatives* sur l'efficacité du programme. Ainsi, il n'est même pas garanti que la version parallèle d'un programme soit au moins aussi rapide que la version séquentielle, quel que soit le langage utilisé. Lors du développement de programmes parallèles, les intuitions sur la performance à l'exécution d'un code parallèle peuvent être totalement fausses. Il est beaucoup plus fiable de faire reposer l'analyse de codes parallèles sur des bases formelles en utilisant des méthodes mathématiques pour la modélisation et l'analyse des performances des programmes.

Lors de l'utilisation de méthodes d'analyse quantitative avant d'implémenter une application, il est impossible d'obtenir des données exactes sur le temps d'exécution des fonctions sur les processeurs. Même lorsqu'une implémentation du système est disponible et utilisable, le temps d'exécution mesuré d'une fonction (par opposition à sa complexité asymptotique) change suivant la charge du processeur sur lequel elle s'exécute, la taille des données à traiter et autres facteurs. Pour ces raisons, le formalisme de modélisation approprié doit utiliser des variables aléatoires probabilistes pour estimer les temps d'exécution. Un tel modèle aboutit à un processus stochastique qui est analysé pour donner un aperçu de la performance éventuelle du système. Lorsque la seule estimation connue du temps d'exécution des fonctions est la durée moyenne, la distribution de nombres aléatoires appropriée est la distribution exponentielle. Les processus stochastiques sur un espace d'états fini pour lesquelles tous les taux de transition entre les états suivent une distribution exponentielle sont une classe importante de processus stochastiques, les chaînes de Markov à temps continu (CTMCs - Continuous Time Markov Chains).

Des méthodes de résolution efficaces existent pour trouver la distribution des probabilités stationnaires d'une CTMC, ce qui rend ces dernières utiles pour la modélisation et l'analyse de systèmes. La distribution stationnaire d'une CTMC est un vecteur exprimant la probabilité que le système soit dans chaque état lorsqu'il atteint un équilibre. L'équilibre est obtenu après une période de "chauffe" du système, une fois que l'influence de l'état initiale ne se fait plus sentir. Des mesures significatives sur les performances du système, telles que le débit, peuvent être calculées à partir de la distribution stationnaire. Cette dernière peut aussi servir à déterminer les goulots d'étranglement du système, les composants sur ou sous utilisés, ou bien encore des verrous vivants (le système atteint un sous-ensemble de son espace d'états, et ne peut plus en sortir).

Traditionnellement, les CTMCs ne sont pas utilisées directement comme un langage formel de modélisation. Elles sont utilisées par le biais d'un langage de modélisation de haut niveau, et la

CTMC est générée en appliquant la sémantique de ce langage. Les formalismes les plus connus sont les réseaux de Petri stochastiques [1], les réseaux d'automates stochastiques [32, 15] et les algèbres de processus stochastiques [22]. L'algèbre de processus pour l'évaluation des performances PEPA [22] est un exemple particulier d'un tel langage de modélisation de haut niveau, dans lequel la sémantique formelle du langage est définie grâce à une sémantique opérationnelle dans le style de Plotkin [33]. Cette définition de la sémantique permet d'obtenir un graphe de transitions étiqueté, qui est généré à partir d'une composition parallèle de processus PEPA séquentiels (appelés composants PEPA). Les étiquettes du graphe sont de la forme (activité, taux), où l'activité est le nom symbolique d'un événement (tel qu'une arrivée ou un départ d'un pipeline), et le taux est le paramètre de la distribution exponentielle associée à l'activité. Une CTMC peut être générée directement à partir du graphe en supprimant les noms des activités.

Dans la suite, nous décrivons comment modéliser les squelettes algorithmiques *Pipeline* et *Donne* à l'aide de PEPA. De tels modèles peuvent être alors utilisés pour évaluer les performances de n'importe quelle application à base de squelettes, après un éventuel re-paramétrage du modèle. La section 3 décrit un simulateur pas à pas pour modèles PEPA, et son utilisation est illustrée dans la section 4 à travers un exemple basé sur les modèles de squelettes décrits précédemment. Enfin, nous présentons les travaux connexes et nous concluons.

2. Modélisation de squelettes algorithmiques

Dans cette section, nous donnons un aperçu de la programmation parallèle structurée et des modèles de performance. Ensuite, nous montrons comment nous combinons les deux approches en créant des modèles de performances de squelettes algorithmiques.

2.1. Une introduction aux squelettes algorithmiques

2.1.1. Programmation parallèle structurée

Les environnements de programmation simple (Posix threads, MPI) sont universels. Ils peuvent être utilisés pour décrire des interactions entre différentes activités, arbitrairement complexes et déterminées dynamiquement. La programmation se fait en choisissant et en combinant des opérateurs extraits d'un petit ensemble simple.

Il est difficile pour le programmeur de comprendre le schéma global de l'application (si un tel schéma existe) en examinant le programme statiquement, et de le changer. Il est très difficile d'implémenter des techniques d'optimisation (statiques et/ou dynamiques) qui concernent plus d'une seule instance de primitive.

Cependant, en pratique, les programmes parallèles ne sont pas constitués de schémas d'interaction dynamiques et arbitraires. Dans la plupart des cas le schéma est prédéterminé. Dans d'autres cas, le non déterminisme est contraint dans un schéma plus large. La programmation parallèle non structurée empêche l'utilisateur d'exprimer des informations sur le schéma général, qui reste implicite lors de l'utilisation de primitives simples.

L'approche structurée du parallélisme propose d'abstraire des schémas classiques de calcul et d'interaction sous forme d'une librairie de fonctions paramétrées, de constructeurs, etc. Le programmeur peut alors exprimer explicitement le fait que l'application suit un ou plusieurs de ces schémas. Ce concept est important car il donne aux développeurs de codes parallèles les outils conceptuels abordant les problèmes qui rendent difficile une programmation parallèle efficace et correcte.

Une introduction plus détaillée du sujet peut être trouvée dans [11, 12].

On dénomme squelette algorithmique [8, 35, 11] une construction de programme qui abstrait un tel schéma de processus et d'interactions. Le programmeur invoque un ou plusieurs squelettes pour décrire la structure du programme, et il les personnalise avec les types et les opérations propres à l'application. Le code gérant l'interaction et l'invocation des opérations dépend implicitement du squelette choisi. Nous rappelons brièvement le concept du *Pipeline* qui est d'utilité reconnue dans différentes applications. Puis nous présentons le squelette *Donne*, souvent utilisé imbriqué dans une étape de pipeline.

2.1.2. Le squelette Pipeline

Dans la forme la plus simple de pipeline [9], une séquence de données en *entrée* est traitée successivement par N_e *étapes*, produisant une séquence de données en *sortie* (Fig. 1). Chaque donnée traverse toutes les étapes dans l'ordre, et le traitement d'une donnée en entrée commence dès que la première étape a terminé de traiter la donnée précédente. Le parallélisme est introduit car plusieurs données sont traitées en même temps dans les différentes étapes du pipeline. Le temps mis à traiter une donnée particulière est donc allongé. Le gain de performance est observé uniquement lorsque plusieurs données sont traitées en même temps dans le pipeline.

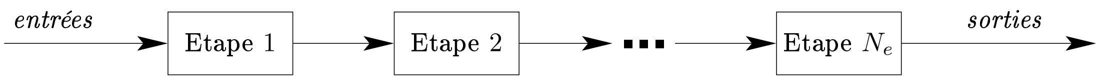

FIG. 1 – Pipeline

2.1.3. Le squelette Donne

Il est bien connu que la performance d'un pipeline dépend du temps de calcul requis par l'étape la plus longue. Ceci peut être amélioré en exploitant le parallélisme à l'intérieur d'une étape, en plus du parallélisme entre les différentes étapes propre au pipeline. Une approche simple consiste à implémenter l'étape avec plusieurs processeurs (travailleurs) qui acceptent les données chacun leur tour, retournant les résultats en conservant l'ordre des données. Le parallélisme intrinsèque à l'étape permet ainsi de pallier au long temps mis pour traiter chaque donnée (sauf la première) dans l'étape (Fig. 2). Ce schéma est représenté par le squelette Donne [10]. Le nom est choisi par analogie au fait de distribuer des cartes (données en entrée) à un groupe de joueurs (les travailleurs) en suivant une stricte rotation, classiquement appelé la donne. Dans notre modèle, on considère Nt_s *travailleurs* pour un squelette Donne s.

Du point de vue utilisateur, il est important de noter que cette approche marche uniquement pour les étapes n'ayant pas d'état interne. Si l'étape maintient et met à jour un état interne lors du traitement consécutif des données, une parallélisation interne plus sophistiquée doit être envisagée.

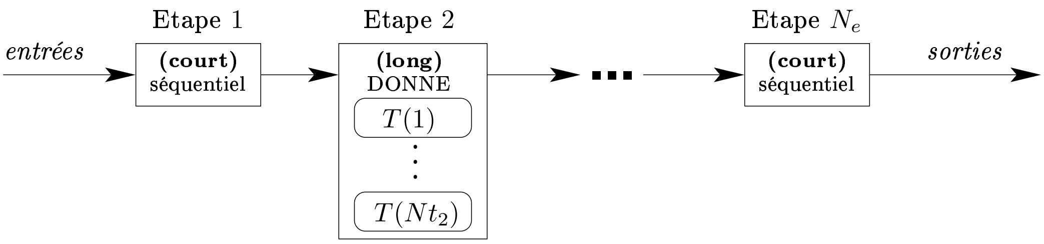

FIG. 2 – Pipeline et Donne

2.1.4. Travaux connexes

Différentes approches à la programmation parallèle structurée ont été développées ces dernières années, et nous donnons ici un aperçu de quelques unes. De nombreux travaux ont été inspirés par la programmation fonctionnelle, ou sont même des extensions de langages fonctionnels.

Ainsi, **P3L** [29, 3, 4] (langage de programmation parallèle de Pise) a été créé pour aider à concevoir des applications parallèles en utilisant des squelettes. Dans ce langage, les calculs parallèles sont réduits à un ensemble de constructeurs. Le prototype **OcamlP3l** [14] regroupe un langage fonctionnel, Objective Caml, avec les squelettes P3L, formant la base d'un environnement de programmation parallèle puissant.

D'autres approches ont été développées sur les fondations de langages classiques, comme la **librairie BSMLlib** [16] pour la programmation en Bulk Synchronous Parallel avec le langage fonctionnel Objective Caml. Elle est basée sur une extension du lambda calcul avec des opérations parallèles sur une structure de donnée (vecteur parallèle).

Une extension du langage fonctionnel paresseux Haskell [30] a abouti à **Eden** [6], un langage fonctionnel parallèle qui donne une nouvelle perspective sur la programmation parallèle. Il laisse suffisamment de contrôle aux programmeurs pour implémenter les algorithmes parallèles efficacement, et en même temps les libère de la gestion bas niveau des processus. Une librairie de squelettes prédéfinis a été ajoutée au langage pour permettre une programmation parallèle facile.

Enfin, d'autres approches sont moins liées à la programmation fonctionnelle. La librairie de squelettes d'Edimbourg **eSkel** [9, 10] est une librairie C construite au dessus de MPI (Message Passing Interface, [20]). Elle permet au programmeur C d'utiliser des squelettes génériques et flexibles pour abstraire les schémas classiques de parallélisme. Dans la suite, nous appliquons nos travaux aux squelettes Pipeline et Donne comme ils sont définis dans cette librairie, étant donné que c'est la librairie que nous développons. Cependant, nous gardons à l'esprit le fait que notre approche peut être étendue à tout autre environnement de squelettes.

Après avoir donné un aperçu de la programmation parallèle structurée et présenté certains squelettes, nous nous concentrons sur l'évaluation des performances, pour pouvoir par la suite introduire des modèles de performances de squelettes.

2.2. Une introduction à l'évaluation des performances

Dans cette section, nous présentons tout d'abord l'algèbre de processus PEPA (Performance Evaluation Process Algebra, [22]). Ensuite nous discutons de quelques outils pour l'évaluation des performances.

2.2.1. Une introduction à PEPA

Le langage PEPA offre un ensemble de combinateurs. Ils permettent de construire les termes du langage définissant le comportement des composants PEPA à travers les activités qu'ils entreprennent et les interactions entre composants. Une information de temps est associée à chaque activité. Ainsi, lorsqu'elle est autorisée, une activité $a = (\alpha, r)$ a lieu et sa durée suit une distribution exponentielle de paramètre r. Si plusieurs activités sont autorisées en même temps, soit en compétition soit indépendamment, on suppose qu'il existe une condition de concurrence (*race condition*) entre elles. Les combinateurs de composants utilisés dans les modèles de squelettes, ainsi que leurs noms et interprétations, sont présentés officieusement ci-dessous.

Préfixe : Le mécanisme de base pour décrire le comportement du système consiste à assigner au composant la première action qu'il doit exécuter en utilisant le combinateur préfixe, dénoté par un point. Par exemple, le composant $(\alpha, r).S$ effectue l'activité (α, r), avec une durée suivant une distribution exponentielle de paramètre r, puis se comporte comme le composant S.

Choix : Le combinateur de choix, dénoté par le symbole $+$, exprime la possibilité de compétition entre différentes activités. Ainsi, le composant $P+Q$ représente un système qui peut se comporter soit comme P, soit comme Q : les deux activités de ses composants sont autorisées. La première activité à finir détermine le choix ; l'autre activité est annulée. Le système se comportera comme le dérivé résultant de l'évolution du composant ayant terminé son activité en premier.

Constante : Il est pratique de pouvoir assigner des noms à des schémas de comportement associés aux composants. Les constantes sont les composants dont le comportement est défini par une équation.

Hiding : Le combinateur de cache *Hiding*, dénoté par P/L, donne la possibilité d'abstraire certains aspects du comportement d'un composant. L'ensemble L des identificateurs des activités visibles détermine les activités qui sont considérées comme internes ou privées pour le composant, et qui apparaissent alors comme une activité de nom inconnu τ.

Coopération : En PEPA, l'interaction directe, ou *coopération* entre composants, est la base de la composition. Ce combinateur est dénoté par $P \underset{L}{\bowtie} Q$, où P et Q sont les deux composant coopérants. L'ensemble utilisé en indice du symbole de coopération, l'*ensemble de coopération* L, détermine les activités sur lesquelles les deux composants sont forcés de se synchroniser. Pour les activités qui ne sont pas dans L, les composants se comportent de façon indépendante, et les activités autorisées simultanément dans les deux composants sont en concurrence. Cependant, une activité qui est dans L ne peut pas avoir lieu avant que les deux composants autorisent cette activité. Les deux composants peuvent alors travailler ensemble pour compléter l'activité partagée. Le taux d'une telle activité peut être modifié pour refléter le travail fourni par chaque composant pour compléter l'activité (pour plus de détails, se référer à [22]). Nous dénotons par $P \parallel Q$ une abréviation de $P \underset{L}{\bowtie} Q$ dans le cas où L est l'ensemble vide.
Dans certains cas, lorsqu'une activité a lieu en coopération entre deux composants, l'un des composants peut être *passif* en ce qui concerne cette activité. Cela signifie que le taux de cette activité n'est pas spécifié (noté $\top$), et il est déterminé au moment de la coopération par le taux de l'activité dans l'autre composant. Toutes les activités passives doivent être synchronisées dans le modèle final.

Le comportement dynamique d'un modèle PEPA est représenté par l'évolution de ses composants, dictée par la sémantique opérationnelle des termes PEPA (c.f. [22]). Ainsi, comme dans les algèbres de processus classiques, la sémantique de chaque terme est donnée à travers un système multi-transitions étiqueté (les multiplicités des arcs sont significatives). Dans le système de transitions, un état correspond à un terme syntaxique du langage, ou *dérivative*, et un arc représente une activité qui provoque l'évolution d'une dérivative vers une autre. L'ensemble des états atteignables s'appelle l'*ensemble des dérivatives*, et ils forment les noeuds du graphe de dérivation. Ce graphe est obtenu en appliquant les règles de la sémantique de façon exhaustive.

Le graphe de dérivation est la base de la chaîne de Markov à temps continu sous-jacente, qui est utilisée pour obtenir des mesures de performances à partir d'un modèle PEPA. Le graphe est systématiquement réduit à une forme dans laquelle il peut être traité comme le diagramme états-transitions de la chaîne de Markov sous-jacente. Le taux de transition entre deux dérivatives P et Q du graphe de dérivation est le taux avec lequel le système passe d'un comportement décrit par le composant P à un comportement décrit par Q. C'est la somme des taux des activités étiquetant les arcs qui connectent les noeuds P et Q.

2.2.2. Outils pour les modèles de performance

Un certain nombre de méthodes et d'outils sont disponibles pour analyser les modèles PEPA [7], ayant pour but d'en apprendre plus à l'utilisateur sur les systèmes modélisés. Un aperçu de ces outils est schématisé dans la figure 3.

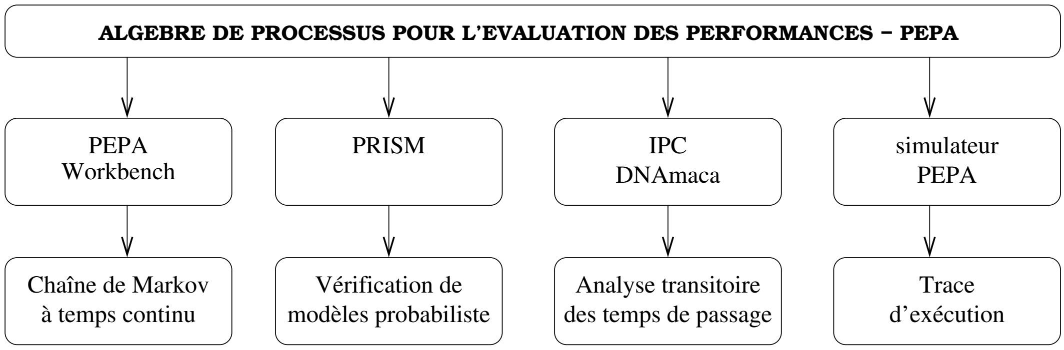

FIG. 3 – Outils pour l'algèbre de processus PEPA

Un des moyens de résoudre directement un modèle PEPA consiste à utiliser la boite à outil *PEPA Workbench* [18] pour générer l'espace d'états du modèle et le générateur infinitésimal de la chaîne de Markov sous-jacente. Le PEPA Workbench peut alors calculer la distribution des probabilités stationnaires du système, et en déduire des mesures de performance telles que le débit et le taux d'utilisation.

Le PEPA Workbench représente l'espace d'états du modèle sous la forme d'une matrice creuse. Une approche différente pour représenter le générateur infinitésimal de la CTMC est celle prise par le vérificateur de modèles symbolique probabiliste PRISM [25], qui stocke les matrices en utilisant des diagrammes de décision binaires multi-terminaux (MTBDDs). Ces derniers sont un format de stockage compact pour les très grands modèles possédant des propriétés de symétries. PRISM accepte PEPA comme un de ses langages d'entrée, et il offre de nombreux algorithmes de solution numérique : méthode de la puissance, Jacobi, Gauss-Seidel, JOR et SOR. De plus, PRISM permet de donner des formules en logique stochastique continue pour vérifier le modèle exprimé par la CTMC. Il fournit ainsi une méthode pour effectuer une vérification personnalisée des performances, dans le but de répondre à des questions combinant le temps, les probabilités, et les chemins à travers l'exécution du système.

Le compilateur *Imperial PEPA Compiler* IPC [5] offre d'autres solutions. Ce programme est écrit dans le langage de programmation fonctionnel paresseux Haskell [30]. Son rôle consiste à compiler un modèle PEPA dans le langage accepté en entrée par l'analyseur de Knottenbelt DNAmaca [24]. Les méthodes de solution offertes par DNAmaca incluent des méthodes directes (élimination de Gauss, Grassmann), des méthodes itératives classiques (Gauss-Seidel, SOR fixe et dynamique), des techniques basées sur le sous-espace de Krylov (BiCG, CGNR, CGS, BiCGSTAB, BiCGSTAB2, TFQMR) et des méthodes basées sur la décomposition (incluant AI (Agrégation-Isolation), AIR). La principale caractéristique ici est que ces méthodes peuvent être utilisées pour calculer des temps de passage pour des modèles PEPA. De telles expressions sont souvent trouvées dans des accords sur la qualité de service (QoS). DNAmaca trace ses résultats comme une fonction de distribution des probabilités ou une fonction cumulative de densité.

Enfin, le comportement du modèle peut être exploré interactivement en utilisant notre nouveau simulateur pas à pas (Section 3). Le PEPA Workbench permet également de déboguer par étapes, mais son utilisation est limitée car il requiert une génération de l'espace d'états du modèle avant de commencer le débogage. Cela signifie qu'il ne peut pas être utilisé pour trouver des synchronisations manquantes (l'espace d'états est alors très large). Notre nouvelle implémentation en Objective Caml n'a pas cette restriction, elle stocke uniquement l'état courant du modèle.

2.3. Modèles des squelettes Pipeline et Donne

Nous présentons les modèles des squelettes algorithmiques Pipeline et Donne de la librairie *eSkel*, décrits en Section 2.1.2 et 2.1.3. Les modèles sont en algèbre de processus PEPA, comme décrit dans la section 2.2.1.

Pour modéliser une application pipeline, nous décomposons le problème selon les étapes, les processeurs et le réseau. Certaines des étapes peuvent alors être modélisées comme un squelette Donne.

Les étapes
La première partie du modèle représente l'application, elle est indépendante des ressources sur lesquelles l'application s'exécute. L'application consiste en N_e étapes, modélisées par un composant PEPA $Etape_e$ $(e = 1..N_e)$.

• Lorsque $Etape_e$ n'est pas une Donne, elle s'exécute séquentiellement. Tout d'abord, elle obtient une donnée (activité $transfert_e$), puis la traite $(traite_{e,1})$, et enfin transfert la donnée au stage suivant $(transfert_{e+1})$. Pour l'activité $traite_{e,1}$, le 1 dans l'indice indique qu'il s'agit du premier (et seul) travailleur pour cette étape (le nombre de travailleurs pour cette étape $Nt_e = 1$). Cet indice prend de l'importance lorsque l'étape est une Donne. La définition PEPA correspondante est dans la figure 4a. Tous les taux sont indéfinis, notés par le symbole $\top$, car les temps de transfert et de traitement dépendent des ressources sur lesquelles l'application tourne. Ils seront définis plus tard, dans une autre partie du modèle.

• Lorsque $Etape_e$ est une Donne, Nt_e travailleurs traitent la séquence de données. Dans notre modèle, nous forçons l'allocation cyclique des données aux travailleurs en introduisant, pour chaque Donne, un composant *Source* et un composant *Puits* qui jouent le rôle d'interface entre les travailleurs et les activités de transfert reliant une étape à ses voisines dans le pipeline. Chaque travailleur $i \in \{1, \ldots, Nt_e\}$ commence par obtenir une donnée de la source avec une activité $importe_{e,i}$, traite la donnée $(traite_{e,i})$, puis exporte le résultat vers le puits $(exporte_{e,i})$. Comme précédemment, les taux des activités *importe* et *exporte* ne sont pas spécifiés, ils seront définis plus loin. On obtient les définitions $Source_e$, $Puits_e$ et $Travailleur_{e,i}$ de la figure 4b, où les travailleurs sont définis pour $i = 1, \ldots, Nt_e$. Tous les travailleurs sont indépendants, et ils sont synchronisés à la source et au puits grâce aux actions *importe* et *exporte*. Nous définissons $LI_e = \{importe_{e,i}\}_{i\in\{1,\ldots,Nt_e\}}$ et $LE_e = \{exporte_{e,i}\}_{i\in\{1,\ldots,Nt_e\}}$ dans la définition de $Etape_e$ (Fig. 4b).

Une fois les étapes définies, l'application est une coopération des différentes étapes par les activités $transfert_e$, pour $e = 2..N_e$. Les activités $transfert_1$ et $transfert_{N_e+1}$ représentent respectivement l'arrivée d'une donnée dans l'application, et la sortie d'un résultat du pipeline. Elles ne représentent pas de transfert de donnée entre étapes, donc ne synchronisent pas le pipeline. La définition du composant Pipeline se trouve dans la figure 4c.

Les processeurs
L'application est exécutée sur un ensemble de N_p processeurs. Chaque travailleur est implémenté par un unique processeur, mais un processeur peut héberger plusieurs travailleurs. Pour conserver la simplicité du modèle, nous intégrons les informations propres au processeur (charge, nombre de travailleurs implémentés, ...) directement dans le taux $\mu_{e,i}$ des activités $traite_{e,i}$ (définies pour les composant $Etape_e$ et $Travailleur_{e,i}$). Chaque processeur est alors représenté par un composant PEPA ayant un comportement cyclique, traitant séquentiellement les données par différents travailleurs. Quelques exemples suivent.

– Lorsqu'il n'y a pas de Donne, lorsque $N_p = N_e$, on affecte un travailleur par processeur :

$$Processeur_e \stackrel{def}{=} (traite_{e,1}, \mu_{e,1}).Processeur_e$$

– Si plusieurs travailleurs sont hébergés par le même processeur, on utilise un choix PEPA. Dans l'exemple suivant ($N_p = 2$, $N_e = 2$, et la première étape est une Donne avec deux travailleurs),

a. Etape sans Donne

$$Etape_e \stackrel{def}{=} (transfert_e, \top).(traite_{e,1}, \top).(transfert_{e+1}, \top).Etape_e$$

b. Etape avec Donne

$$\begin{array}{rcl}
Source_e & \stackrel{def}{=} & (transfert_e, \top).(importe_{e,1}, \top).\\
& & (transfert_e, \top).(importe_{e,2}, \top).\\
& & \ldots\\
& & (transfert_e, \top).(importe_{e,Nt_e}, \top).Source_e\\
Travailleur_{e,i} & \stackrel{def}{=} & (importe_{e,i}, \top).(traite_{e,i}, \top).(exporte_{e,i}, \top).Travailleur_{e,i}\\
Puits_e & \stackrel{def}{=} & (exporte_{e,1}, \top).(transfert_{e+1}, \top).\\
& & (exporte_{s,2}, \top).(transfert_{e+1}, \top).\\
& & \ldots\\
& & (exporte_{e,Nt_e}, \top).(transfert_{e+1}, \top).Puits_e\\
Etape_e & \stackrel{def}{=} & Source_e \underset{LI_e}{\bowtie} (Travailleur_{e,1} \mid\mid \ldots \mid\mid Travailleur_{e,Nt_e}) \underset{LE_e}{\bowtie} Puits_e
\end{array}$$

c. L'application Pipeline

$$Pipeline \stackrel{def}{=} Etape_1 \underset{\{transfert_2\}}{\bowtie} Etape_2 \underset{\{transfert_3\}}{\bowtie} \cdots \underset{\{transfert_{N_e}\}}{\bowtie} Etape_{N_e}$$

d. Les processeurs

$$Processeurs \stackrel{def}{=} Processeur_1 \mid\mid Processeur_2 \mid\mid \ldots \mid\mid Processeur_{N_p}$$

e. Le réseau

$$\begin{array}{rrlcl}
Reseau \stackrel{def}{=} & & (transfert_1, \lambda_1).Reseau & + \ldots + & (transfert_{N_e+1}, \lambda_{N_e+1}).Reseau\\
& + & (importe_{e,1}, \lambda I_{e,1}).Reseau & + \ldots + & (importe_{e,Nt_e}, \lambda I_{e,Nt_e}).Reseau\\
& + & (exporte_{e,1}, \lambda E_{e,1}).Reseau & + \ldots + & (exporte_{e,Nt_e}, \lambda E_{e,Nt_e}).Reseau
\end{array}$$

f. Modèle final

$$Modele \stackrel{def}{=} Reseau \underset{L_r}{\bowtie} Pipeline \underset{L_p}{\bowtie} Processeurs$$

FIG. 4 – Définitions PEPA

le premier processeur exécute le premier travailleur de chaque étape, et le second processeur ne s'occupe que du deuxième travailleur de la première étape (étape qui est donc distribuée sur les deux processeurs).

$$\begin{array}{rcl} Processeur_1 & \stackrel{def}{=} & (traite_{1,1}, \mu_{1,1}).Processeur_1 + (traite_{2,1}, \mu_{2,1}).Processeur_1 \\ Processeur_2 & \stackrel{def}{=} & (traite_{1,2}, \mu_{1,2}).Processeur_2 \end{array}$$

Une fois les processeurs définis individuellement au cas par cas, et étant donné que tous les processeurs sont indépendants, on peut définir l'ensemble des processeurs comme une composition parallèle des différents composants $Processeur$ (Fig. 4d).

Le réseau
Plutôt que de représenter directement la structure physique de l'architecture du réseau, notre modèle de réseau sert à définir les taux des actions logiques de communication ($transfert, importe, exporte$) du modèle de l'application, en utilisant des informations du "Network Weather Service" NWS [36], en particulier la latence des communications inter-processeurs. La définition du modèle est immédiate une fois les taux des activités définis : λ_e pour $transfert_e$, $\lambda I_{e,i}$ pour $importe_{e,i}$, et $\lambda E_{e,i}$ pour $exporte_{e,i}$. Par exemple, si seule l'étape e est une Donne, on obtient la définition de la figure 4e. Si d'autres étapes sont également des Donnes, on doit ajouter les activités $importe$ et $exporte$ correspondantes dans le choix. Nous ne détaillons pas dans cet article le calcul des taux, qui n'est pas significatif pour ces travaux.

Modèle général
Une fois les différentes composantes du modèle définies, le modèle général est une coopération des étapes, des processeurs et du réseau. On utilise pour cela les ensembles de coopération L_r pour synchroniser l'application et le réseau (ensemble des activités $transfert$, $importe$ et $exporte$), et L_p synchronise l'application et les processeurs (ensemble des activités $traite$). La définition se trouve dans la figure 4f.

3. Description du simulateur

Les modèles de haut niveau de systèmes complexes ne sont pas forcément corrects du seul fait qu'ils ont été exprimés dans un langage formel en théorie bien fondé. Comme tout programme, ils peuvent contenir des erreurs qui doivent être corrigées. Les erreurs dans des programmes sont parfois identifiées clairement comme appartenant à des classes connues (division par zéro, débordement de tableau, et autres). Parfois, les erreurs sont plus difficiles à détecter (ne retourne pas la bonne valeur entière, chaîne de caractères mal formatée, etc...). Ces erreurs doivent être détectées en comparant les valeurs incorrectes retournées et les valeurs attendues.

De la même façon, les erreurs dans les modèles de haut niveau de processus stochastiques sont parfois triviales, et parfois non. Dans la première catégorie, on trouve les interblocages et les synchronisations pour lesquelles aucun des partenaires de la synchronisation ne définit le taux de l'activité. De telles erreurs sont détectées durant la génération de l'espace d'états, si ce n'est pas avant. Les erreurs du deuxième type incluent la spécification de processus de Markov ayant trop (ou pas assez) d'états ou de transitions, et des erreurs de protocole lorsque des activités sont déclenchées dans le mauvais ordre par un composant. Ces erreurs sont bien plus difficiles à détecter car rien n'est signalé lors de la génération de l'espace d'états ou lors de la résolution du modèle pour calculer la distribution stationnaire. Comme pour les programmes retournant une mauvaise valeur, une erreur logique peut être détectée en comparant l'espace d'états générés avec celui que l'on attend. Malheureusement, nous ne disposons pas d'un espace d'états attendu auquel comparer nos résultats. De telles erreurs sont donc très difficiles à détecter, et il n'y a pas de solution facile dans ce cas.

Pour aider partiellement la résolution de ce problème, nous avons implanté un simulateur de modèles PEPA qui donne la possibilité au programmeur d'évoluer pas à pas dans l'espace d'états du système, en cherchant des synchronisations manquantes, des composants non atteignables, ou autres erreurs de modélisation. Le simulateur est implanté en enveloppant le langage PEPA dans Objective Caml, et en utilisant la boucle de haut niveau O'Caml pour passer des commandes servant à générer les dérivatives de l'état courant du modèle, à choisir les activités que l'on veut suivre, et à dérouler pas à pas le modèle ou à mettre des points d'arrêts sur des composants ou des activités et faire tourner le modèle jusqu'à ce qu'ils soient rencontrés. Ceci peut aider le modeleur à trouver des activités ou des composants non atteignables, mettant à jour un problème dans le modèle.

Ce simulateur est complètement indépendant du simulateur pas à pas implémenté dans le PEPA Workbench. L'utilisation du langage applicatif O'Caml permet une vérification rigoureuse des types, et nous permet d'implémenter bien plus de fonctionnalités que celles disponibles dans le simulateur précédent. De plus, il n'est plus nécessaire de générer l'espace d'états du modèle.

Le code du nouveau simulateur est lié au modèle que l'on considère, car les activités et les taux sont définis comme un type O'Caml pour permettre une vérification rigoureuse des types. Nous détaillons le code d'un tel simulateur pour un modèle de squelette dans la section suivante. Le coeur du simulateur est évidemment le même pour chaque modèle, mais il est nécessaire d'inclure directement les parties dépendantes du modèle dans le code. Nous voulons attirer l'attention du lecteur sur le fait que l'un des objectifs du simulateur sera d'avoir une génération automatique du code dépendant du modèle, pour permettre une utilisation immédiate. Même si cette génération automatique n'a pas encore été implantée, nous montrons dans la section suivante comment un tel simulateur fonctionne sur un exemple spécifique.

4. Exemple d'application

Dans cette section, nous illustrons l'utilisation du nouveau simulateur O'Caml pour les modèles de squelettes présentés en section 2.3. Comme nous l'avons déjà remarqué, une partie du code du simulateur dépend du modèle et doit donc être écrite à la main, mais nous projetons de le générer automatiquement avec un compilateur de modèles PEPA. Le but de cette section est de montrer comment le simulateur marche à travers un exemple.

On considère un pipeline à deux étapes, où la première étape est une Donne avec deux travailleurs. La première partie du simulateur, dépendante du modèle, définit les identificateurs utilisés dans le modèle, ainsi que les activités et les taux. Ils correspondent à la définition du modèle donnée en section 2.3. Nous devons ajouter l'activité spéciale `Tau` pour permettre le "hiding" PEPA. Le type taux permet de définir les taux symboliques apparaissant dans le modèle, qui sont utilisés de préférence aux valeurs réelles pour simplifier la correction du modèle. Le taux spécial `Top` est ajouté pour les besoins de synchronisation, il correspond au symbole $\top$ dans PEPA. Le taux `Const` permet d'assigner directement une valeur sans utiliser de nom symbolique.

```
type identificateur = Source | Trav1 | Trav2 | Puits | Etape1 | Etape2
                    | Reseau | Proc1 | Proc2 | Procs;;
type activite   = Tau of activite | Importe1 | Importe2 | Exporte1 | Exporte2
       | Transfert1 | Transfert2 | Transfert3 | Traite11 | Traite12 | Traite2;;
type taux       = Top | Const of float | Lambda | LambdaI | LambdaE | Mu;;
```

On définit alors le type composant fidèle à la sémantique de PEPA (et indépendant du modèle). La vérification des types sur le modèle se fait automatiquement à partir de cette définition.

```
type composant  = Prefixe of ((activite * taux) * composant)
                | Choix   of composant * composant
```

```
                | Coop    of composant * activite list * composant
                | Hiding  of composant * activite list
                | Var     of identificateur;;
```

Une autre partie du simulateur propre au modèle est alors la définition du modèle lui-même, qui consiste en une fonction définissant chaque identificateur. Nous présentons seulement une partie de cette définition vu qu'elle est très proche du modèle auquel on se réfère.

```
let definition = function
    Source -> Prefixe ((Transfert1, Top), Prefixe ((Importe1, Top),
              Prefixe ((Transfert1, Top), Prefixe ((Importe2, Top), Var Source))))
  | Trav1  -> Prefixe ((Importe1, Top),   Prefixe ((Traite11, Top),
                        Prefixe ((Exporte1, Top), Var Trav1)))
  | Etape1 -> Coop (Var Source, [Importe1;Importe2], Coop ((Coop (Var Trav1, [],
                    Var Trav2)),[Exporte1;Exporte2],Var Puits))
  | Etape2 -> Prefixe ((Transfert2, Top),  Prefixe ((Traite2, Top),
                                           Prefixe ((Transfert3, Top), Var Etape2)))
  | Reseau -> Choix (Choix (Choix (Choix (Choix (Choix (
                     Prefixe ((Transfert1, Lambda),  Var Reseau),
                     Prefixe ((Importe1,   LambdaI), Var Reseau)),
                     Prefixe ((Exporte1,   LambdaE), Var Reseau)), ...))))
  | Proc1  -> Choix (Prefixe ((Traite11,Mu), Var Proc1),
                     Prefixe ((Traite2, Mu), Var Proc1))
  | Procs  -> Coop  (Var Proc1, [], Var Proc2)
  | ...  ;;
```

Le simulateur consiste alors en une série de définitions et de fonctions permettant l'affichage des composants. Nous pouvons aussi assigner des valeurs réelles aux noms symboliques définis précédemment. Le coeur du simulateur est la fonction `derive_un_pas`, qui retourne la liste des dérivatives d'un composant passé en argument, ainsi que la liste des activités et taux permettant d'atteindre ces dérivatives.

```
val derive_un_pas: composant -> ((activite * taux) * composant) list = <fun>
```

L'utilisation de cette fonction permet à l'utilisateur de localiser à la main les problèmes, et de garantir que tout correspond à ses espérances. Les erreurs évidentes dans la définition du modèle sont détectées automatiquement par O'Caml grâce à une vérification rigoureuse des types.

Pour utiliser le simulateur, il faut définir le composant que l'on veut analyser. Dans notre cas, l'application pipeline est définie par

```
let pipeline = Coop(Var Procs, [Traite11;Traite12;Traite2],
               Coop(Coop(Var Etape1,[Transfert2],Var Etape2),[Transfert1;Transfert2;
                     Transfert3;Importe1;Importe2;Exporte1;Exporte2],Var Reseau));;
```

On peut alors afficher le résultat de la fonction `derive_un_pas`, qui indique la liste des activités pouvant avoir lieu et le taux associé, ainsi que le composant dérivé. Dans cet exemple, la seule activité autorisée est `Transfert1`, avec un taux `Lambda`.

```
# print_string (affiche_derivatives (derive_un_pas pipeline));;
-(transfert1, lambda)-> (Processeurs <traite1, traite2> ((((importe1,top).
(transfert1, top).(importe2,top).Source <importe1,importe2> ((Trav1 <> Trav2)
```

```
<exporte1,exporte2> Puits)) <transfert2> Etape2) <transfert1, transfert2,
transfert3, importe1,importe2, exporte1, exporte2> Reseau));
- : unit = ()
```

Il est alors possible d'évoluer dans le système en dérivant les composants obtenus en résultat pas à pas. Remarquons qu'en ce qui concerne nos modèles de squelettes, différentes paramétrisations du modèle ne devrait pas affecter la structure de la chaîne de Markov sous-jacente. La phase de déboguage est donc nécessaire une seule fois pour chaque modèle de squelette, mais inutile pour chaque nouvelle application utilisant ce squelette. Cependant, lorsqu'on s'intéresse à un nouveau modèle PEPA, un simulateur adéquat doit être automatiquement créé et le simulateur pas à pas peut aider à trouver des problèmes et des erreurs dans le modèle.

5. Travaux connexes et conclusions

De nombreux outils pour l'analyse d'algèbres de processus stochastiques simulent un chemin à travers l'espace d'états du modèle, en suivant la sémantique opérationnelle du langage par étapes. Dans les travaux présents, ainsi que dans les travaux précédents [19], notre but est de faciliter la tâche du programmeur pour trouver des erreurs dans la formulation du modèle. Dans d'autres travaux, les auteurs ont des buts différents, mais ils peuvent être utilisés à la même fin. Souvent, la motivation pour une exécution pas à pas suivant les transitions d'un modèle stochastique sert à simuler le comportement du modèle. Lorsqu'on considère une classe très générale de modèles, avec des variables aléatoires de distribution générale, des délais déterministes ou un comportement non déterministe, il se peut que cette méthode d'analyse soit la seule disponible. Philips et Cardelli [31] présentent une approche basée sur la simulation pour analyser des modèles de calculs de processus stochastiques, sur un modèle de machine abstraite qui est une variante du π-calcul [34]. Dans ces travaux, l'algorithme de Gillepsie est implémenté pour permettre la simulation du modèle du système, générant la trace du chemin d'exécution pendant un intervalle de temps donné. Le simulateur est implémenté en Objective Caml et produit des fichiers en sortie qui peuvent être utilisés par d'autres outils pour visualiser les résultats.

D'autres travaux visent à améliorer les aspects quantitatifs des programmes applicatifs à base de squelettes algorithmiques, mais en utilisant des approches totalement différentes. Par exemple, Alt et Gorlatch [2] proposent une optimisation des mécanismes de Java RMI (Remote Method Invocation) en vue d'améliorer les performances d'applications ciblant les grilles d'ordinateurs. Une approche encore différente est celle de Di Cosmo et Pelagatti [13], motivée par le développement de OcamlP3L. Ils introduisent un calcul décrivant plusieurs stratégies pour distribuer des tableaux sur un ensemble de processeur, et un coût est associé à chaque distribution. Cette approche permet ainsi de prendre des décisions quantitatives pour améliorer les performances des applications. Finalement, Hayashi et Cole [21] ont travaillé sur VEC-BSP, un langage fonctionnel simple qui incorpore les concepts de "forme" [23] et de squelettes parallèles pour permettre la définition d'un calcul des coûts d'exécution, évalués statistiquement. Ce calcul est lui-même paramétré par les mesures de coût de BSP.

Nous avons donné dans ce papier un aperçu de la programmation parallèle de haut niveau et de la notion de squelettes. Nous avons ensuite fait une courte introduction à l'évaluation des performances, à l'aide des algèbres de processus PEPA, et nous avons présenté des outils classiques pour analyser des modèles d'algèbre de processus. Enfin, nous avons décrit comment mélanger les deux sujets en proposant des modèles de performances (en PEPA) de squelettes algorithmiques (de la librairie *eSkel*). Nous avons alors introduit l'idée d'un simulateur pas à pas pour les modèles PEPA. Une première version du simulateur a été implémentée à l'aide du langage fonctionnel O'Caml, permettant une vérification rigoureuse des types et une détection aisée des erreurs dans le modèle. Le simulateur a été testé sur nos modèles de squelettes, mais il peut être utilisé sur n'importe quel autre modèle PEPA.

La prochaine étape de ces travaux consiste à implémenter un compilateur de modèles PEPA qui génère automatiquement le simulateur pour n'importe quel modèle. Ceci sera assez immédiat une fois qu'une analogie détaillée aura été faite entre les définitions PEPA et les définitions O'Caml correspondantes à inclure dans le simulateur. Le coeur du simulateur est commun à tous les modèles, mais la partie spécifique au modèle, incluant les définitions des composants et des activités, doit être ré-écrite pour correspondre à la description PEPA. De plus, d'autres fonctionnalités peuvent être ajoutées au simulateur, par exemple la dérivation de plusieurs étapes, ou bien la dérivation suivant un sous-ensemble d'activités itérée tant que possible. A plus long terme, on projette d'étudier une application impliquant plus de squelettes, pour illustrer la résolution de problèmes réels. De plus, le simulateur sera appliqué à des applications n'étant pas nécessairement à base de squelettes, et cette tâche sera facilitée une fois que la génération automatique du simulateur à partir du modèle sera implémentée.

Références

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, New York, 1995.

[2] M. Alt and S. Gorlatch. A Prototype Grid System Using Java and RMI. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 401–414. Springer Verlag, 2003.

[3] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, 1997.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : A structured high level programming language and its structured support. *Concurrency : Practice and Experience*, 7(3) :225–255, May 1995.

[5] J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Derivation of passage-time densities in PEPA models using IPC : The Imperial PEPA Compiler. In G Kotsis, editor, *Proc. of the 11th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 344–351, University of Central Florida, October 2003.

[6] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden : Language Definition and Operational Semantics. Technical Report 10, Philipps-University of Marburg, 1996.

[7] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA performance modelling tools. *IEE Proceedings—Software*, 146(1) :11–19, February 1999.

[8] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation.* MIT Press & Pitman, ISBN 0-262-53086-4, 1989.

[9] M. Cole. eSkel : The **e**dinburgh **Ske**leton **l**ibrary. Tutorial Introduction. *Internal Paper, School of Informatics, University of Edinburgh*, 2002.

[10] M. Cole. eSkel : The **e**dinburgh **Ske**leton **l**ibrary Version 2.0 – Draft API reference manual. *Internal Paper, School of Informatics, University of Edinburgh*, 2003.

[11] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3) :389–406, 2004.

[12] M. Cole. Why structured parallel programming matters. Keynote address at EuroPar 2004, September 2004. Pisa, Italy.

[13] R. Di Cosmo and S. Pelagatti. A calculus for dense array distributions. *Parallel Processing Letters*, 13(3) :377–388, 2003.

[14] M. Danelutto, R. D. Cosmo, X. Leroy, and S. Pelagatti. Ocamlp3l a functional parallel programming system. Technical Report LIENS-98-1, E.N.S. Paris, 1998.

[15] P. Fernandes. *Méthodes Numériques pour la solution de systèmes Mar koviens à grand espace d'états.* PhD thesis, Institut National Polytechnique de Grenoble, France, 1998.

[16] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[17] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In V. Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 55-68, INRIA, 2004.

[18] S. Gilmore and J. Hillston. The PEPA Workbench : A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in LNCS, pages 353–368, Vienna, May 1994. Springer-Verlag.

[19] S. Gilmore and J. Hillston. Performance modelling in PEPA with higher-order functions. In N. Thomas and J. Bradley, editors, *Proc. of the Sixteenth UK Performance Engineering Workshop*, pages 35–46, Department of Computer Science, The University of Durham, July 2000.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, 2nd edition.* MIT Press, 1999.

[21] Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Parallel Programs. *Parallel Algorithms and Applications*, 17(1) :59–84, 2002.

[22] J. Hillston. *A Compositional Approach to Performance Modelling.* CUP, 1996.

[23] C.B. Jay. Costing Parallel Programs as a Function of Shapes. *Science of Computer Programming*, 37(1-3) :207–224, 2000.

[24] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, 1996.

[25] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM : A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66, 2002.

[26] X. Leroy. Writing efficient numerical code in Objective Caml, July 2002. Web page available at `http://caml.inria.fr/ocaml/numerical.html`.

[27] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *Int. Jo. of Computer and Information Science*, 5(3), 2004.

[28] R. Milner. How ML evolved. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(1), 1983.

[29] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, London, 1998.

[30] S. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[31] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. In Anna Ingólfsdóttir and Hanne Riis Nielson, editors, *Proceedings of the Second Int. Workshop on Concurrent Models in Molecular Biology*, pages 11–25, August 2004. To appear in ENTCS.

[32] B. Plateau. *De l'Evaluation du parallélisme et de la synchronisation.* PhD thesis, Université de Paris XII, Orsay (France), 1984.

[33] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[34] C. Priami. Stochastic π-calculus. In S. Gilmore and J. Hillston, editors, *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*, pages 578–589. Special Issue of *The Computer Journal*, 38(7), December 1995.

[35] F.A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer Verlag, 2002.

[36] R. Wolski, N.T. Spring, and J. Hayes. The network weather service : a distributed resource performance forecasting service for metacomputing. *FGCS*, 15(5–6) :757–768, 1999.