Réseaux FIFO Colorés Stricts pour la formalisation des applications de visualisation scientifique interactives

Abderrahim Ait Wakrime*, Sébastien Limet*, Sophie Robert* [1]

*Univ. Orléans, INSA Centre Val de Loire, LIFO, EA 4022,
F-45067, Orléans, France

Résumé. La programmation par composants est devenue une approche essentielle et très utilisée en génie logiciel. En particulier, dans le cadre des applications de visualisation scientifique interactives, cette approche offre une architecture claire aux développeurs permettant de bien séparer les différentes parties fonctionnelles de l'application comme l'interaction, la simulation et la visualisation. La modélisation d'applications de visualisation scientifique interactive doit permettre la description des comportements de chaque composant et de l'assemblage de ces composants en une application. Cet assemblage s'exprime par un schéma de communications qui peut être très complexe avec la possibilité de perdre des messages pour gagner en performance. Nous proposons un modèle par composants spécifique pour ces applications, associé à une formalisation par des réseaux FIFO colorés. Le modèle a pour objectifs de décrire les différents comportements des composants et du réseau de communications en les formulant afin d'offrir des outils de vérifications.

1 Introduction

Depuis plusieurs décennies, les approches à base de composants ont attiré beaucoup d'attention en génie logiciel. Plusieurs domaines typiques ont connu une réussite en exploitant cette approche comme les services web, les systèmes critiques ou les systèmes distribués.

Le principe général d'une approche par composants (Armstrong et al. (2006); Emmerich et Kaveh (2002)) s'appuie sur un composant logiciel défini comme une unité de composition avec des interfaces contractuellement spécifiées qui lui permettent de communiquer avec l'extérieur. Un composant peut être ainsi utilisé par d'autres logiciels et grâce à une description externe de son utilisation et de ses interfaces peut être instancié dans une application sans connaissance particulière sur son implémentation. Les approches par composants sont utilisées dans de nombreux domaines. Soit les approches sont très générales avec comme objectif de respecter les propriétés fondamentales des approches par composants soit elles sont spécialisées à une classe d'applications (Borgdorff et al. (2013); Goodale et al. (2003); Allard et al. (2004)). En effet, le modèle de composition doit parfois être très spécifique afin d'offrir les moyens d'assembler les composants pour construire une application répondant aux caractéristiques du domaine visé (la performance, des composants très hétérogènes, etc.).

[1]Ce travail est financé par le projet ANR "ExaviZ"

Concevoir une application de visualisation scientifique interactive nécessite des compétences dans des métiers très différents comme le rendu graphique, les interactions et la gestion des périphériques et enfin la simulation ou la modélisation en fonction du domaine scientifique de ces applications (biochimie, médicale, géosciences, etc.). Utiliser une approche par composants pour les construire permet de rendre accessible la visualisation scientifique à des non spécialistes et à des thématiciens grâce à la séparation des préoccupations que ces approches favorisent. Ainsi, de nombreux workflows scientifiques reposent sur une approche par composants (Ludascher et al. (2006); Borgdorff et al. (2013); Goodale et al. (2003)). Mais ces workflows sont très spécifiques à un domaine d'application avec des objets prédéfinis. Pour des applications plus générales de visualisation scientifique, il est nécessaire de définir une approche par composants qui permet des schémas de communications bien particuliers. Par exemple, les applications de visualisation scientifique interactives peuvent impliquer des contraintes sur la cohérence de données face à des schémas de communications basés sur la perte de données pour assurer la performance de l'application. Il est donc nécessaire de maîtriser ses pertes pour assurer la cohérence par exemple des résultats de deux composants de simulations qui coopèrent, résultats qui sont affichés par un composant de rendu.

Dans ce contexte, nous proposons un modèle par composants basé sur un système de coordination exogène similaire à Blair et al. (2009) qui permet de réaliser de la composition temporelle et d'exprimer des schémas de communications et des politiques de synchronisation extérieures aux composants. Notre modèle s'appuie sur un composant qui délivre des services (calcul, rendu ou interaction) à partir de ports qui communiquent leurs données via des connecteurs. Nos connecteurs sont proches des connecteurs de Arbab (2004) mais ils n'utilisent pas un protocole du type *requête-réponse*. Au contraire, pour permettre d'exprimer des communications très asynchrones, notre système de coordination est dirigé par les données et les connecteurs délivrent les données dès que le récepteur est prêt à les consommer. Ainsi l'émetteur peut fonctionner à sa propre fréquence, ce qui favorise des applications performantes.

Nous souhaitons associer à notre modèle par composants une boîte à outils permettant à un utilisateur de concevoir, construire et modifier son application à partir d'une description des modules qui la constituent, de leur assemblage et des contraintes associées. Par exemple, Limet et al. (2011) décrit un élément de cette boîte à outils pour traiter de la construction d'une application sous contraintes de cohérence. Dans ce papier, nous proposons d'intégrer à cette boîte à outils une formalisation de notre modèle qui permettra de vérifier que certaines propriétés d'une application comme son démarrage, sa vivacité ou l'absence d'interblocage sont bien respectées. Ainsi, notre modèle par composants est formalisé par les réseaux FIFO colorés Jensen et Kristensen (2009); Peterson (1981) qui permettent de décrire le comportement dynamique d'un système événementiel grâce à une sémantique formelle. Ces réseaux ont été très utilisés pour modéliser et analyser différents types de processus comme les protocoles, les systèmes de fabrication, des processus métier mais aussi des applications de type services web avec Hamadi et Benatallah (2003) par exemple.

Dans le cadre de la reconfiguration dynamique d'une application, certains travaux Léger et al. (2010); Heinzemann et al. (2012); Dormoy et al. (2011) proposent des étapes de vérification des transformations des applications. Cependant il s'agit d'applications auto-adaptatives et la vérification ne s'appuie pas sur une modélisation du système mais est destinée à valider la faisabilité de la reconfiguration. En général, ces travaux utilisent Fractal Blair et al. (2009) qui définit ses propres éléments de reconfiguration. Cependant, Dormoy et al. (2011) propose une

approche similaire à la nôtre en utilisant la logique temporelle. Mais contrairement aux réseaux de Petri, une formalisation par la logique temporelle n'offre pas un simulateur des applications qu'elle modélise. De plus, dans un modèle data-flow, le jeton représente plus naturellement les données qui transitent dans l'application.

Dans cet article, après avoir décrit notre modèle par composants et sa sémantique dans la section 2, nous définissons dans la section 3 la construction du réseau FIFO coloré et nous montrons que la sémantique de ce réseau respecte bien la sémantique de l'application correspondante. Enfin, en conclusion nous donnons des perspectives à partir d'une spécification plus précise de notre boîte à outils.

2 Le modèle de composants

Une application de visualisation scientifique interactive est basée sur des modules de simulation, d'interaction et de rendu graphique. La diversité de ces modules et leurs différents types de fonctionnement nécessitent de définir une approche par composants spécifique répondant à ces contraintes lors de l'assemblage des composants pour obtenir une application fonctionnelle et performante. Notre approche s'appuie sur des composants, des connecteurs et des liens. Les composants réalisent les différents traitements (simulation, rendu, ...), les connecteurs et les liens permettent l'acheminement des données et ainsi l'interconnexion des composants.

2.1 Les composants

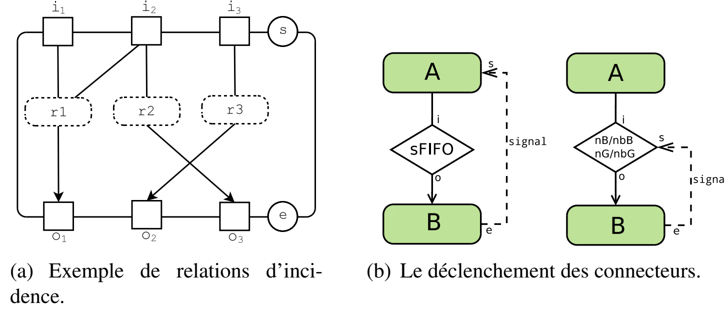
Un composant est une boîte noire formée de ports d'entrée et de ports de sortie qui jouent le rôle d'interfaces entre le composant et l'extérieur. Ces ports ainsi que des relations d'incidence permettent de décrire le composant et ses différents manières de consommer des données sur ses ports d'entrée et de produire des résultats sur ses ports de sortie.

Définition 1 *Un **composant** est un quadruplet* $C = (Id, pIn_C \cup \{s\}, pOut_C \cup \{e\}, RI_C)$.

- Id *son identifiant unique ;*
- pIn_C *l'ensemble de ses ports d'entrée ;*
- $pOut_C$ *l'ensemble de ses ports de sortie avec* pIn_C *et* $pOut_C$ *disjoints ;*
- s *un port d'entrée de déclenchement et* e *un port de sortie de signalement ;*
- RI_C *un ensemble de relations d'incidence ;*

Un composant possède ainsi des ports d'entrée qui utilisent les données produites par d'autres composants et des ports de sortie qui fournissent des données. Les données qui transitent dans l'application sont appelées des messages et un composant ne connaît pas l'origine des messages qu'il reçoit ni la destination des messages qu'il produit. Le comportement du composant est décrit par des relations d'incidence. Comme l'illustre la définition 2, elles expriment des dépendances entre les ports d'entrée et de sortie.

Définition 2 *Une relation d'incidence d'un composant* C *est un couple* $r = \langle RI^{in}, RI^{out} \rangle$ *où* $RI^{in} \subseteq pIn_C$ *est l'ensemble des ports d'entrée de* r *qui s'ils contiennent des données assurent que le composant produit des données sur les ports* $RI^{out} \subseteq pOut_C$ *de* C.



(a) Exemple de relations d'incidence.

(b) Le déclenchement des connecteurs.

FIG. 1 – *Relations d'incidence et connecteurs*

Une relation d'incidence correspond à un mode opérationnel du composant et son comportement se définit alors par l'ensemble des relations d'incidence qui sont vérifiées *c-à-d.* dont les ports d'entrées sont alimentés comme illustrée figure 1(a). On peut alors déterminer quels sont les résultats produits à partir des ports de sortie des relations d'incidence vérifiées.

Dans la suite, nous utilisons les notations suivantes :

- $Port_C$ désigne l'ensemble des ports de données d'entrée et de sortie d'un composant C, $Port_C = pIn_C \cup pOut_C$,
- $RI^{in}(r)$ et $RI^{out}(r)$ désignent respectivement, l'ensemble des ports d'entrée et de sortie de la relation d'incidence r.
- Pour un ensemble de relations d'incidence $\mathcal{E}$, $RI^{in}(\mathcal{E})$ et $RI^{out}(\mathcal{E})$ désignent respectivement, l'ensemble des ports d'entrée et de sortie des relations d'incidence appartenant à $\mathcal{E}$, $RI^{in}(\mathcal{E}) = \bigcup_{r\in\mathcal{E}} RI^{in}(r)$ et $RI^{out}(\mathcal{E}) = \bigcup_{r\in\mathcal{E}} RI^{out}(r)$.
- $RI^{out}(pIn)$ désigne l'ensemble des ports de sortie des règles validées par l'ensemble pIn c-à-d. $RI^{out}(pIn) = \{o \in RI^{out}(r) | RI^{in}(r) \subseteq pIn\}$.

Dans le cadre des applications visées, les composants sont itératifs. Pour notre modèle, le processus itératif d'un composant est une boucle appelée *wait-get-put* définie par :

Définition 3 *L'itération d'un composant appelé* wait-get-put *consiste à*

wait : *Attendre des données sur tous les ports d'au moins une relation d'incidence du composant et attendre un signal de déclenchement sur son port s s'il est connecté.*

get : *Consommer une donnée sur* tous *les ports d'entrée des relations d'incidence vérifiées pour exécuter la tâche correspondante à chacune des relations vérifiées ;*

put : *Envoyer les résultats produits sur les ports de sortie de toutes les relations d'incidence vérifiées et envoyer un signal notifiant la fin d'une itération sur le port e*

Chaque composant gère en interne un numéro d'itération qui permet de suivre ce processus “wait-get-put”. Ce numéro d'itération est également indiqué en entête de chaque message produit par le composant sur les ports de sortie des relations d'incidence vérifiées.

2.2 Les connecteurs et les liens

Assembler des composants, pour réaliser une application, consiste à définir un schéma de communications permettant les échanges de données sur les interfaces des composants soit leurs ports d'entrée et de sortie. Ce schéma de communications se base sur des connecteurs et des liens. Les connecteurs réalisent l'acheminement des messages selon une politique adaptée à la communication entre des composants itératifs ayant des fréquences différentes. Les liens formalisent les liaisons entre composants et connecteurs.

Définition 4 *Un connecteur est un quadruplet* $conn = (Id, \{i, s\}, \{o\}, t)$ *où* i *est un port d'entrée,* o *un port de sortie et* t *son type. les paramètres* Id *et* s *sont similaires à leurs homonymes dans un composant.*

Dans le cadre de la visualisation scientifique nous avons défini trois types de connecteurs tels que décrits dans Limet et al. (2011) et illustré figure 1(b) :

- **sFIFO** est une liaison simple de type file FIFO où, pour éviter les débordements, l'émetteur attend un signal de déclenchement sur son port s en général envoyé par le récepteur. Il est à noter que ce connecteur est défini par le quadruplet $(Id, \{i\}, \{o\}, t)$.
- **bBuffer** et **nbBuffer** conservent les messages arrivant et ne délivrent un message qu'à la réception d'un signal de déclenchement sur leur port s. Le **nbBuffer** est la variante non bloquante du **bBuffer** *c-à-d.* il génère un message vide pour le récepteur lorsqu'il est déclenché alors qu'il n'a aucun message en attente.
- **bGreedy** et **nbGreedy** ne stockent que le dernier message reçu et le délivrent à la réception d'un signal de déclenchement sur leur port s. Le **nbGreedy** est la variante non bloquante du **bGreedy**.

Enfin, les liens permettent de relier composants et connecteurs via leurs ports.

Définition 5 *Un lien est un couple* $\langle x^p, y^q \rangle$ *où* x, y *sont des composants ou des connecteurs,* $p \in pOut_x \cup \{e\}$ *et* $q \in pIn_y \cup \{s\}$*. Lorsque* $p \neq e$*,* $q \neq s$ *et que* x *ou* y *est un connecteur le lien est dit* lien de données*. Lorsque* $p = e$*,* $q = s$ *et que* x *est un composant alors le lien est dit* lien déclencheur*. Deux liens de données* $\langle x_1^{p_1}, y_1^{q_1} \rangle$ *et* $\langle x_2^{p_2}, y_2^{q_2} \rangle$ *sont* compatibles *si* $y_1^{q_1} \neq y_2^{q_2}$.

2.3 Le graphe d'application

Finalement, l'application va être décrite par un graphe qui connecte les ports de sortie aux ports d'entrée des composants et des connecteurs.

Définition 6 *Une application App est définie par le graphe* $(Comp \bigcup Conn, Dl \bigcup Tl)$ *où* $Comp$ *est l'ensemble des composants,* $Conn$ *l'ensemble des connecteurs,* Dl *l'ensemble des liens de données deux à deux compatibles et* Tl *les liens déclencheurs.*

Il faut remarquer que dans une application, un port d'entrée de données ne peut être alimenté que par un seul lien. Cette restriction peut être levée en définissant des connecteurs supplémentaires définissant comment les messages sont alors traités (concaténation, séquentialisation, etc.).

Définition 7 *Dans une application* $(Comp \bigcup Conn, Dl \bigcup Tl)$*, le port* p *dans un composant* x *est dit connecté s'il est le sommet d'un des arcs de* $Dl \bigcup Tl$*.*

Dans une application, les composants doivent être connectés de façon à assurer que les données arrivant sur un port d'entrée pourront être consommées par au moins une relation d'incidence d'une part et qu'un port de sortie connecté fournira bien des données d'autre part. La définition suivante formalise ces notions.

Définition 8 *Notons* pIn_C^c *(resp.* $pOut_C^c$*) l'ensemble des ports d'entrée (resp. de sortie) connectés pour un composant* C *dans une application App. On dit alors qu'une relation d'incidence* r *est déclenchable si* $RI^{in}(r) \subseteq pIn_C^c$*. Notons* $\mathcal{RI}_C^c \subseteq \mathcal{RI}_C$ *l'ensemble des relations d'incidence qui peuvent être déclenchées à partir de* pIn_C^c *alors*

- *un composant est* bien connecté en entrée *si* $RI^{in}(\mathcal{RI}_C^c) = pIn_C^c$*,*
- *un composant est* bien connecté en sortie *si* $RI^{out}(\mathcal{RI}_C^c) \subseteq pOut_C^c$*.*
- *Une application* App *est* bien formée *si tous ses composants sont bien connectés en entrée et en sortie.*

Une application *bien formée* permet de prévenir les cas de saturations des ports d'entrée de relations d'incidence qui ne sont jamais déclenchés et les cas de blocage dûs à la connexion d'un port de sortie de ces mêmes relations d'incidence.

L'application figure 2 est basée sur une simulation de dynamique moléculaire. Dans cette application, le composant *Simu* génère des positions d'atomes affichés en temps réel par le composant *Visu*. De plus, un composant *Interaction* (par exemple gérant un Omni Phantom®) permet de contrôler la position 3D d'un avatar évoluant parmi les atomes. Il permet aussi d'activer une force calculée par un composant *ForceGenerator* à partir d'une distance entre l'avatar et l'atome le plus proche, force renvoyée à la simulation pour influencer la dynamique moléculaire. Lorsque la force est appliquée, le composant *Simu* génère également l'énergie du système d'atomes, énergie qui peut être affichée par *Visu*.

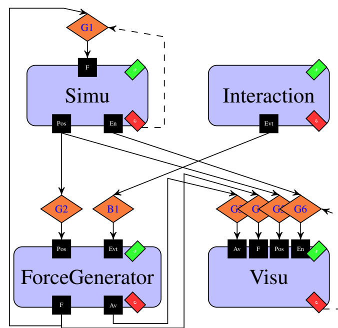


FIG. 2 – *Exemple de graphe d'application (seuls deux liens déclencheurs ont été représentés).*

Afin de garantir la performance, le schéma de communications est basé sur des connecteurs de type *Greedy* afin de perdre les données qui pourraient saturer les composants les plus

lents. Cependant, les événements émis par *Interaction* qui déclenchent le calcul de la force influençant le cours de la simulation sont de type clic souris et ne peuvent donc pas être perdus. D'où un connecteur de type *Buffer* entre *Interaction* et *ForceGenerator*. Le choix entre des connecteurs de type bloquant ou non est fortement lié aux relations d'incidence des composants. Par exemple, les relations d'incidence de *ForceGenerator* montrent que le port Evt peut être qualifié de bloquant car le composant ne procède à une itération que si ce port est alimenté. Donc, en fonction de la capacité de *ForceGenerator* à traiter des messages vides, nous choisirons un *bBuffer* ou un *nbBuffer*, le second permettant de préserver la performance du module. En revanche, le port Pos n'étant pas dans les deux relations d'incidence, est dit non bloquant et le connecteur $G2$ peut être un *bGreedy* sans que cela influence la fréquence de *ForceGenerator*. Il en est de même pour les composants *Simu* et *Visu* qui n'ont aucun port bloquant. Les connecteurs en amont peuvent être simplement des connecteurs *bGreedy* qui ont l'avantage d'être moins coûteux en ne générant pas de messages vides.

Notons que même si ce n'est pas utilisé sur cet exemple, le port s est toujours un port bloquant et permet de contrôler, s'il est connecté, les itérations du composant quelque soit ses relations d'incidence. Notons également que si l'utilisateur impose qu'au niveau de l'affichage les différentes données soient cohérentes, par exemple les positions *pos* des atomes affichés sont les mêmes que les positions utilisées pour calculer la force par *ForceGenerator*, il sera nécessaire de définir un nouveau réseau de communication qui, en particulier, maîtrise les pertes des greedy en amont de *Visu*. Dans Limet et al. (2011), pour une approche par composants avec une relation d'incidence par composants nous avons décrit une construction d'un schéma de communications sous contraintes de cohérence basées sur les numéros d'itérations des composants.

2.4 La sémantique du modèle

Nous souhaitons modéliser notre approche par des réseaux de Petri. Il est donc nécessaire de formaliser la sémantique des composants, des connecteurs et d'une application afin ensuite de démontrer que notre modélisation est bien conforme à cette sémantique.

Pour un composant et un connecteur il s'agit de formaliser l'opération effectuée lorsqu'ils sont déclenchés. Pour cela nous devons introduire des définitions supplémentaires pour exprimer la consommation et la production des messages. Dans ces définitions, les composants et les connecteurs se déclenchent uniquement en fonction de l'état de leurs ports d'entrée. Ensuite il s'agit de définir comment ils mettent à jour leur état après le déclenchement. L'application, par l'intermédiaire des liens, va elle aussi mettre à jour l'état de ces objets en vidant les ports de sortie des uns pour remplir les ports d'entrée des autres.

Nous aurons besoin de la notion de file d'attente (FIFO) avec les opérations d'ajout de e à une file FIFO f notée f+e, d'accès au prochain élément à sortir de $f \neq \emptyset$ (le plus ancien) notée $f \uparrow$ et de la suppression du plus ancien élément notée f--.

Définition 9 *Soit* $\overrightarrow{P} = (p_i)_{1 \leq i \leq n}$ *un vecteur de ports distincts et* $\overrightarrow{M} = (m_i)_{1 \leq i \leq m}$ *un vecteur de messages. On dit que* $\overrightarrow{M}$ *et* $\overrightarrow{P}$ *sont compatibles s'ils ont la même longueur et s'ils sont ordonnés de telle manière que le message* m_i *est destiné au port* p_i*. Le message* m_i *peut être noté vide* $m_i = \emptyset$ *pour signifier l'absence de message.*

Pour un vecteur de messages $\overrightarrow{M}$ compatible avec $\overrightarrow{P}$, la notation m_p (où p est le port p_i de $\overrightarrow{P}$) désigne le message m_i de $\overrightarrow{M}$.

Définition 10 *Soit $\overrightarrow{pIn_C}$ le vecteur des ports d'entrée d'un composant C et $\overrightarrow{M}$ un vecteur de messages compatible, on dit que $\overrightarrow{M}$ vérifie $r \in RI_C$ ssi $\forall p \in RI^{in}(r)\ m_p \neq \emptyset$. On note $V_C(\overrightarrow{M})$ l'ensemble des relations d'incidence de C vérifiées par $\overrightarrow{M}$.*

Un composant, selon notre modèle, procède à une itération dès que les ports d'au moins une de ses relations d'incidence sont alimentés. Les messages correspondants sont alors consommés et les autres messages qui peuvent ne pas être vides attendent la prochaine itération. Enfin les résultats produits sont portés sur les ports de sortie des relations d'incidence ayant déclenché l'itération. Soient Φ_C et Ψ_C les deux fonctions ci-dessous qui permettent de définir la consommation et la production des messages lorsqu'un composant procède à son itération.

Définition 11 *Soit Φ_C la fonction qui, à partir d'un vecteur de messages en entrée d'un composant compatible avec les ports d'entrée, construit le vecteur de messages restant en entrée à la fin de l'itération.*

$$\Phi_C(\overrightarrow{M}) = \overrightarrow{M'} \text{ tel que } m'_p = \left| \begin{array}{ll} \emptyset & \forall p \in RI^{in}(V_C(\overrightarrow{M})) \\ m_p & \text{sinon} \end{array} \right.$$

Définition 12 *Soit Ψ_C la fonction qui, à partir d'un vecteur de messages $\overrightarrow{M}$ en entrée d'un composant et compatible avec les ports d'entrée, construit le vecteur de messages $\overrightarrow{M'}$ portés sur les ports de sortie.*

$$\Psi_C(\overrightarrow{M}) = \overrightarrow{M'} \text{ tq } \overrightarrow{M'} \text{ compatible avec } \overrightarrow{POut_C} \text{ et } m'_p = \left| \begin{array}{ll} \omega_p & \text{si } p \in \cup_{r \in V_C(\overrightarrow{M})} RI^{out}(r) \\ \emptyset & \text{sinon} \end{array} \right.$$

où ω_p est le résultat produit par C à destination du port p.

Soient $\overrightarrow{M^{In}}$ un vecteur de messages compatible avec $\overrightarrow{pIn_C}$, $\overrightarrow{M^{Out}}$ un vecteur de messages compatible avec $\overrightarrow{POut_C}$, it le numéro d'itération du processus itératif du composant et m_e, m_s les messages associés aux ports de déclenchement et de signalement, l'état d'un composant est défini par le tuple $(\overrightarrow{M^{In}}, \overrightarrow{M^{Out}}, it, m_e, m_s)$. L'itération d'un composant est alors formalisée par le passage de l'état $(\overrightarrow{M^{In}}, \overrightarrow{M_\emptyset}, it, \emptyset, 1)$ à l'état $(\Phi(\overrightarrow{M^{In}}), \Psi(\overrightarrow{M^{In}}), it+1, 1, \emptyset)$ lorsque $V_C(\overrightarrow{M^{In}}) \neq \emptyset$ ($\overrightarrow{M_\emptyset}$ désigne un vecteur ne contenant aucun message).

Il est à noter que si le port s d'un composant n'est pas connecté dans l'application, on considère que m_s est remis à 1 après chaque itération. Le déclenchement du composant ne dépend alors que de la vérification d'au moins une relation d'incidence.

La sémantique des connecteurs est plus simple à définir. Le connecteur *sFIFO* peut s'assimiler à un fil qui laisse transiter les messages sachant qu'il ne peut recevoir qu'un seul message à la fois. Si l'état du connecteur est décrit par (m_{in}, m_{out}) où m_{in} est le message sur son port i et m_{out} est le message porté sur son port o, *sFIFO* passe de l'état $(m_{in}, \emptyset)$ à $(\emptyset, m_{in})$.

Pour les deux autres familles, il faut définir la gestion des messages reçus avec stockage pour les connecteurs de type *buffer* ou avec perte pour les connecteurs de type *greedy*.

L'état d'un connecteur de type *greedy* est décrit par un tuple $(m_{in}, m_{out}, m_{st}, m_s)$ où m_{in} est le message sur son port i, m_{out} le message porté sur o, m_{st} le message conservé par le connecteur et m_s le message associé au port s du connecteur. Le déclenchement sera différent

bGreedy		bBuffer	
E_d	E_f	E_d	E_f
$(m_{in}, \emptyset, \mathbf{m_{st}}, 1)$	$(m_{in}, \mathbf{m_{st}}, \emptyset, \emptyset)$	$(m_{in}, \emptyset, \mathbf{L_{st}}, 1)$	$(m_{in}, \mathbf{L_{st}}\uparrow, \mathbf{L_{st}}\text{--}, \emptyset)$
$(\mathbf{m_{in}}, \emptyset, m_{out}, \emptyset)$	$(\emptyset, \emptyset, \mathbf{m_{in}}, \emptyset)$	$(\mathbf{m_{in}}, \emptyset, L_{st}, \emptyset)$	$(\emptyset, \emptyset, L_{st}+m_{in}, \emptyset)$
nbGreedy		**nbBuffer**	
E_d	E_f	E_d	E_f
$(m_{in}, \emptyset, \mathbf{m_{st}}, 1)$	$(m_{in}, \mathbf{m_{st}}, \emptyset, \emptyset)$	$(m_{in}, \emptyset, \mathbf{L_{st}}, 1)$	$(m_{in}, \mathbf{L_{st}}\uparrow, \mathbf{L_{st}}\text{--}, \emptyset)$
$(\mathbf{m_{in}}, \emptyset, \emptyset, 1)$	$(\emptyset, m_{\emptyset}, \emptyset, \emptyset)$	$(m_{in}, \emptyset, \emptyset, 1)$	$(m_{in}, m_{\emptyset}, \emptyset, \emptyset)$
$(\mathbf{m_{in}}, \emptyset, m_{st}, \emptyset)$	$(\emptyset, \emptyset, \mathbf{m_{in}}, \emptyset)$	$(\mathbf{m_{in}}, \emptyset, L_{st}, \emptyset)$	$(\emptyset, \emptyset, L_{st}+\mathbf{m_{in}}, \emptyset)$

TAB. 1 – *Sémantique des connecteurs (E_d état de déclenchement, E_f état final)*

suivant que le connecteur est bloquant ou non. Le message vide (qui est différent de l'absence de message) est noté $m_{\emptyset}$. Les comportements sont résumés par les deux premiers sous tableaux du tableau 1 où les messages en gras indiquent la présence obligatoire d'une donnée.

Pour les connecteurs de type *buffer* ce n'est plus le dernier message qui est stocké mais tous les messages qui arrivent sur le port i. Ainsi l'état du connecteur est représenté par le tuple $(m_{in}, m_{out}, L_{st}, m_s)$ où m_{in} est le message sur son port i, m_{out} le message porté sur o, L_{st} la file FIFO des messages contenus dans le buffer de ces connecteurs et m_s le message porté sur le port s. Les deux derniers sous tableaux du tableau 1 synthétisent le nouvel état du connecteur lorsqu'il est déclenché soit à l'arrivée d'un signal $m_s = 1$ sur le port s ou à l'arrivée d'un message sur i. Comme précédemment, les messages écrits en gras indiquent la présence effective d'une donnée. Cette convention est étendue pour la file FIFO L_{st}.

Finalement, la sémantique d'une application peut être définie de proche en proche grâce à la sémantique des composants et des connecteurs. La construction d'une application consiste à assembler les composants et les connecteurs par des liens. Ces liens représentent de simples arêtes dans le graphe et peuvent être considérés comme des fils laissant passer les messages les uns après les autres. Ils agissent donc comme des files FIFO.

Soit une application App dont le graphe est $(Comp \bigcup Conn, Dl \bigcup Tl)$, pour définir son état global, on associe à chaque lien $l \in Dl \bigcup Tl$ la file d'attente $\vec{l}$. L'état de App est donc l'union des états de chaque connecteur, de chaque composant et des états de chaque file d'attente associée aux liens. Ainsi, lorsque l'application est dans un état donné, elle va passer dans l'état suivant en appliquant une des règles de déclenchement d'un des objets de notre modèle, connecteur, composant ou lien. Ces règles de déclenchement modifient l'état courant de l'objet déclenché (pour le lien c'est la file d'attente qui est modifiée) mais également l'état d'autres objets de la manière suivante :

- A la fin d'une étape d'un connecteur (resp. d'un composant) C, un message porté sur un port de sortie va alimenter les files d'attente de tous les liens $\langle C^o, C'^i \rangle$ avec $o \in pOut_C$, $C' \in Comp\ (resp.\ C' \in Conn)\ et\ i \in pIn_{C'}$.

- Suite à la mise à jour de la file d'attente d'un lien de données $\langle c^o, C^i \rangle$ $c \in Comp\ (resp.\ c \in Conn)$, $o \in pOut_c$, $C \in Conn$ (resp. $C \in Comp$ et $i \in pIn_C$) va être mis à jour puisque m_{in} est modifié (resp. $\overrightarrow{M^{In}}$ est modifié).

- Le message $m_s = 1$ est porté sur un port de déclenchement s d'un connecteur ou d'un composant C, l'état de C est alors modifié, lorsque tous les liens $\langle c^e, C^s \rangle \in Tl\ avec\ c \in Comp$ ont des files d'attente contenant au moins un signal.
- Après le déclenchement d'un connecteur (resp. d'un composant) C, l'état de tous les liens $\langle C^o, c^i \rangle$ où $o \in pOut_C$, $c \in Comp$ (resp. $c \in Conn$) avec $i \in pIn_c$ est mis à jour.
- Après le déclenchement d'un composant C, l'état de tous les liens $\langle C^e, c^s \rangle$ où $c \in Conn$ sont mis à jour.

3 Modélisation en Réseaux FIFO Colorés

Dans cette section, nous allons montrer comment formaliser notre modèle de composants par une classe de réseaux de Petri appelée réseaux FIFO colorés stricts (abrégé en $sCFN$). Pour démontrer cette formalisation, la sémantique des $sCFN$ sera également comparée aux sémantiques que nous avons présentées dans la section précédente.

3.1 Réseaux FIFO Colorés stricts

Les réseaux FIFO ont étés introduits par Finkel et Memmi (1982). Dans ce type de réseaux de Petri les places sont les files d'attentes. Une version hybride pour le réseau de Petri colorés a été introduit par Benalycherif et Girault (1996); Jensen et Kristensen (2009). Dans cette classe particulière, certaines places peuvent être des files d'attente de type FIFO d'autres non. Pour notre approche par composants, l'ordre des messages doit être préservé. Par conséquent toutes les places de nos réseaux sont des files FIFO, de plus nous imposons qu'un seul jeton soit consommé ou produit sur chaque arc. Il s'agit d'une restriction des réseaux FIFO colorés que nous appelons réseaux FIFO colorés stricts (abrégé en $sCFN$).

Un jeton est couvert par une valeur de données, qui appartient à un type donné. Un arc joignant une place et une transition est étiqueté par une expression qui est une fonction de couleur déterminant comment un jeton est consommé par une place ou produit dans une place lors du franchissement d'une transition. Ainsi les expressions arc décrivent comment l'état du $sCFN$ change lorsque les transitions se produisent.

Formellement, un réseau FIFO coloré strict est un tuple $sCFN = (P, T, A, \Sigma, V, C, E, I)$ où :

- P, T, A définissent la structure du réseau avec :
  - P un ensemble non vide et fini de places qui sont des files d'attente de type FIFO ;
  - T un ensemble non vide et fini de transitions, avec $P \cap T = \emptyset$ et nous affectons pour chaque $t \in T$ un poids ;
  - A un ensemble non vide et fini d'arcs.
- Σ, V sont utilisés pour définir les types et les variables :
  - Σ un ensemble fini et non vide de types, nommés également couleurs ;
  - V un ensemble de variables typées par un élément de Σ.

- C, E, I sont définis afin de réaliser les inscriptions dans le réseau :
  - $C : P \rightarrow \Sigma$ est une fonction de couleur, elle associe à chaque place p une couleur de jeton. Chaque jeton sur p doit avoir le type $C(p)$;
  - E est une fonction qui associe à chaque arc a une expression (appelée *arc-expression*) du type de la place de l'arc a ;
  - I est une fonction d'initialisation qui associe à certaines places un jeton du type de la place.

Pour formaliser l'évolution d'un $sCFN$, on rappelle qu'une substitution σ de domaine V où V est un ensemble fini de variables typées est une application qui associe à chaque élément x de V une expression du type de x. L'état d'un $sCFN$ n est défini par une fonction appelée *marquage* qui indique le contenu de chacune des places du $sCFN$. Une transition $t \in T$ est *activée* pour un marquage M avec la substitution σ si pour tout arc de la forme $\langle p, t\rangle$ de A on a $M(p) \neq \emptyset$ et $M(p)\uparrow = \sigma(E(\langle p, t\rangle))$. En d'autres termes une transition est activée si toutes ses places d'entrée p ne sont pas vides et que le prochain élément de leur file d'attente est compatible avec l'arc-expression de l'arc $\langle p, t\rangle$. L'*application* de la transition va consister à consommer le jeton le plus ancien de chacune des places d'entrée de la transition et à ajouter un nouveau jeton dans chacune des places de sortie de la transition. Plus formellement, pour un $sCFN$ n, soit $t \in T$, M un marquage et σ une substitution tels que t est activée pour M avec σ, on note $Eat^n_{t,\sigma}(M)$ le marquage M' tel que

$$M'(p) = \left| \begin{array}{l} M(p)\text{-- } \forall p \in P \text{ t.q. } \langle p, t\rangle \in A \\ M(p) \text{ sinon} \end{array} \right.$$

et l'*application de la transition* t est notée $Next^n_{t,\sigma}(M)$, c'est le marquage M'' tel que

$$M''(p) = \left| \begin{array}{ll} Eat^n_{t,\sigma}(M) + \sigma(e) & \forall p \in P \text{ t.q. } \langle t, p\rangle \in A \text{ et } E(\langle t, p\rangle) = e \\ Eat^n_{t,\sigma}(M) & \text{sinon} \end{array} \right.$$

Le marquage initial d'un $sCFN$ n est le marquage M tel que

$$M(p) = \left| \begin{array}{l} I(p) \text{ si } \exists p \mapsto e \in I \\ \emptyset \text{ sinon} \end{array} \right.$$

Une transition $t \in T$ du $sCFN$ n est *activable* pour un marquage M s'il existe une substitution σ telle que t est activée pour M avec σ. L'évolution d'un $sCFN$ consiste à appliquer successivement des transitions activables à partir d'un marquage M. Lorsque plusieurs transitions ayant des places d'entrée communes sont activables pour un marquage M, la transition avec le plus grand poids est appliquée.

La formalisation de notre approche par composants en $sCFN$ consiste à définir séparément le $sCFN$ des composants, des connecteurs puis enfin à combiner ces $sCFN$ pour construire celui de l'application. Pour automatiser ces constructions nous avons besoin de trois structures supplémentaires comme illustrés dans la figure 3. Il s'agit d'exprimer les opérations suivantes :

- La duplication de données ou de signaux permettant de reproduire un signal ou un message vers plusieurs destinations. Comme illustré figure 3(a), la transition t_1 est activable si la place P_1 contient un jeton représentant le message ou le signal à dupliquer. La transition t_1 consomme alors ce jeton et l'envoie dupliqué vers toutes ses places de sortie ;

- La synchronisation sert à produire un seul signal à partir de plusieurs signaux. Comme illustré figure 3(b) lorsque toutes les places P_1, P_2 et P_3 sont marquées, la transition t_1 est activée. Elle consomme alors ces jetons et envoie un seul jeton de même type vers la seule place de sortie ;

- L'incrémentation du compteur de l'itération du composant est exprimée par le $sCFN$ illustré figure 3(c). Ainsi si la place P_1 est marquée par un jeton de type entier, la transition t_1 est activable. Elle reçoit alors l'entier, l'incrémente et envoie la dernière valeur calculée sous forme d'un jeton vers la place P_1.

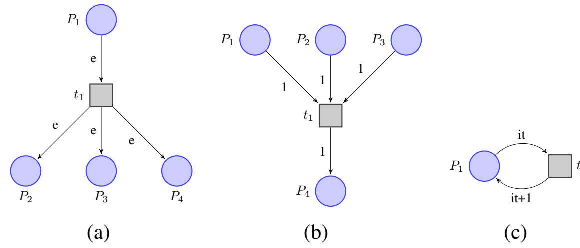


FIG. 3 – *Structures sCFN de la duplication (a), la synchronisation (b) et l'itération (c).*

3.2 Modèle des composants en sCFN

L'idée générale de la transformation est la suivante : les ports (d'entrée et de sortie) du composant seront représentés par des places, et chaque comportement du composant sera modélisé par une transition. Comme déjà vu dans la section 2.1, un composant effectue une itération dès que tous les ports d'entrée d'une de ses relations d'incidence contiennent des messages et toutes les relations d'incidence applicables sont déclenchées lors d'une itération. Pour traduire ce comportement en $sCFN$, nous allons considérer toutes les combinaisons de déclenchements possibles du composant et créer une transition pour chacune de ces combinaisons. Ces transitions auront pour poids le nombre de places en entrée afin d'être sûr de déclencher toutes les relations d'incidence applicables. Étant donné que nous souhaitons raisonner sur une notion de cohérence basée sur le numéro d'itération associé à chaque message, nous allons avoir besoin de définir un compteur permettant de simuler ces numéros d'itération. Ce compteur sera implémenté par une place particulière et utilisera la structure définie 3(c).

L'ensemble des comportements possibles d'un composant C se définit à partir de l'ensemble des parties non vides de RI_C (noté $\mathcal{P}(RI_C)$) par $B_C^{in} = \{RI^{in}(E)|E \in \mathcal{P}(RI_C)\}$. On peut noter que deux parties différentes de RI_C peuvent avoir le même ensemble de ports d'entrée.

Remarque 1 *On peut noter que deux parties différentes de RI_C peuvent avoir le même ensemble de ports d'entrée, par exemple si $RI_C = \{r_1, r_2\}$ avec $r_1 = \langle\{i_1, i_2\}, \{o_2\}\rangle$ et $r_2 = \langle\{i_2\}, \{o_1\}\rangle$, les ensembles $\{r_1\}$ et $\{r_1, r_2\}$ ont les mêmes ports d'entrée. Cela signifie simplement que lorsque r_1 est déclenchable alors r_2 l'est aussi et donc nous allons produire*

une seule transition avec comme entrées i_1 et i_2 dans le réseau de Petri associé au composant. Cette transition aura en sortie o_1 et o_2.

Dans les algorithmes suivants la notation $AddArc(a, e, A, E)$ où a est un arc, e une expression d'arc, A un ensemble d'arcs et E un ensemble d'expressions d'arc, sera un raccourci pour $A := A \cup \{a\}$; $E := E \cup \{a \mapsto e\}$; La fonction de transformation d'un composant C est définie par l'algorithme 1. L'application de cet algorithme au composant de la remarque 1 est illustrée dans la figure 4.

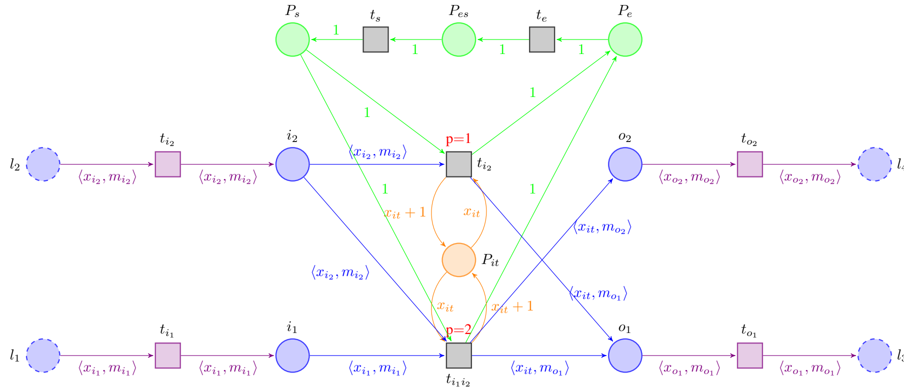


FIG. 4 – *sCFN(C) pour le composant de la remarque 1. Les places l_1, l_2, l_3, l_4 ainsi que les transitions t_{i_1}, t_{i_2}, t_{o_1} et t_{o_2} ne font pas partie de $sCFN(C)$ mais représentent comment les ports du composant sont reliés à l'extérieur.*

Nous allons montrer que le réseau $sCFN(C)$ simule bien le comportement du composant C tel que défini dans la section 2.4.

Pour un marquage M, nous allons noter $P_{in}(\vec{M}(C))$ le vecteur de messages compatible avec pIn_C tel que

$$P_{in}(\vec{M}(C))|_p = \left| \begin{array}{l} M(p)\uparrow \text{ si } M(p) \neq \emptyset \\ \emptyset \text{ sinon} \end{array} \right.$$

On définit de la même manière $P_{out}(\vec{M}(C))$.

Soit M un marquage tel que $M(P_s) = 1$, $M(P_{it}) = n$, $M(P_e) = M(P_{es}) = \emptyset$. On ne fait pas d'hypothèse sur les places qui représentent les ports de sortie car étant des files FIFO, les messages qui y seront stockés seront tous restitués dans leur ordre d'arrivée dans le réseau, nous n'avons donc pas besoin de nous assurer que ces places soient vides pour commencer une itération. Par contre on peut constater que, par construction, aucune transition de $sCFN(C)$ ne sera activable si $P_{in}(\vec{M}(C))$ ne valide aucune relation d'incidence de C. Par contre, si $P_{in}(\vec{M}(C))$ valide au moins une relation d'incidence alors, d'après la sémantique des $sCFN$, la transition activable ayant le plus fort poids sera appliquée (*c-à-d.* celle qui simule toutes les relations validées par $P_{in}(\vec{M}(C))$). L'application de cette transition va enlever un message dans chacune des places qui représentent des ports d'entrée des relations validées et placera un nouveau message estampillé du numéro d'itération courant dans toutes les places

Entrée : C un composant
Sortie : $(P, T, A, \Sigma, V, C, E, I)$ $sCFN$

```
// Une place par port, une place pour le numéro
   d'itération et une place pour le changement d'itération
```
$P := \{P_p | p \in Port_C\} \cup \{P_{it}, P_e, P_s, P_{es}\}$;
$T := \emptyset$; $A := \emptyset$; $E := \emptyset$;
$\Sigma := \{Int\} \cup \{Type(p) | p \in Port_C\}$;
```
// Une variable de type message, une variable qui
   représente le numéro d'itération associé au message et
   une variable pour le numéro d'itération du composant
```
$V := \{x_p, m_p | p \in Port_C\} \cup \{x_{it}\}$;
```
// Les places des ports ont des messages du type du port
```
$C := \{P_p \mapsto Type(p) | p \in Port_C\}$;
```
// Les autres places contiennent un entier
```
$C := C \cup \{P_{it} \mapsto Int, P_e \mapsto Int, P_s \mapsto Int, P_{es} \mapsto Int\}$
```
// Ajout d'une transition pour chaque comportement
```
Pour chaque $B \in B_C^{in}$ **Faire**
- $T := T \cup \{t_B\}$;
- // P_s doit contenir le signal pour activer la transition
- $AddArc(\langle P_s, t_B\rangle, 1, A, E)$;
- // Un message consommé sur chaque port d'entrée de B
- **Pour chaque** $i \in B$ **Faire**
  - $AddArc(\langle P_i, t_B\rangle, \langle x_i, m_i\rangle, A, E)$;
- // Un message produit sur chaque port de sortie de $RI^{out}(B)$
- **Pour chaque** $o \in RI^{out}(B)$ **Faire**
  - $AddArc(\langle t_B, P_o\rangle, \langle x_{it}, m_o\rangle, A, E)$;
- // Incrémentation du numéro d'itération
- $AddArc(\langle P_{it}, t_B\rangle, x_{it}, A, E)$;
- $AddArc(\langle t_B, P_{it}\rangle, x_{it} + 1, A, E)$;
- // Envoi du signal de fin d'itération
- $AddArc(\langle t_B, P_e\rangle, 1, A, E)$;

```
// Ajout des transitions de changement d'itération
```
$T := T \cup \{t_e, t_s\}$;
$AddArc(\langle p_e, t_e\rangle, 1, A, E)$;
$AddArc(\langle t_e, p_{es}\rangle, 1, A, E)$;
$AddArc(\langle p_{es}, t_s\rangle, 1, A, E)$;
```
// On initialise toutes les places des ports de données
   par la liste vide, la place gérant le numéro
   d'itération à 0 et enfin la place d'activation à 1.
```
$I := \{P \mapsto [] | P \notin \{P_{it}, P_e, P_s\}\} \cup \{P_{it} \mapsto 0, P_s \mapsto 1\}$;
```
// Le poids de chaque transition est le nombre de place
   d'entrée de la transition
```

Algorithme 1 : La fonction $sCFN$ pour un composant

qui représentent les ports de sortie des relations validées. L'application de la transition va aussi ajouter 1 à la place P_{it} et placer le signal 1 dans le port P_e. Après l'application de cette transition, aucune transition représentant les comportements ne pourra être appliquée car la place P_s est vide. La seule transition activable est t_e qui va placer le signal dans la place P_{es} (cette transition permet aussi de dupliquer ce message si le port e est connecté dans l'application où est utilisé le composant). Puis la transition t_s va replacer le signal dans la place P_s (cette transition permettra de synchroniser la place P_s avec l'arrivée de signaux venant d'autres éléments de l'application si le port s est connecté). Ce cycle est bien conforme au comportement du composant défini dans la section 2.4.

3.3 Modèle des connecteurs en sCFN

Pour relier des composants, nous utilisons des connecteurs qui gèrent la politique de gestion des messages dans le canal de communication. La transformation du connecteur se réalise, comme pour un composant, en représentant les ports par des places et les comportements internes du connecteur par des transitions. Les connecteurs ont deux ports de données (entrée et sortie) et un port de déclenchement s nécessaires pour communiquer avec les autres éléments de l'application, sauf le connecteur *sFIFO* qui a juste deux ports de données (entrée et sortie). Le connecteur *sFIFO* est trivial comme le montre la figure 5(a).

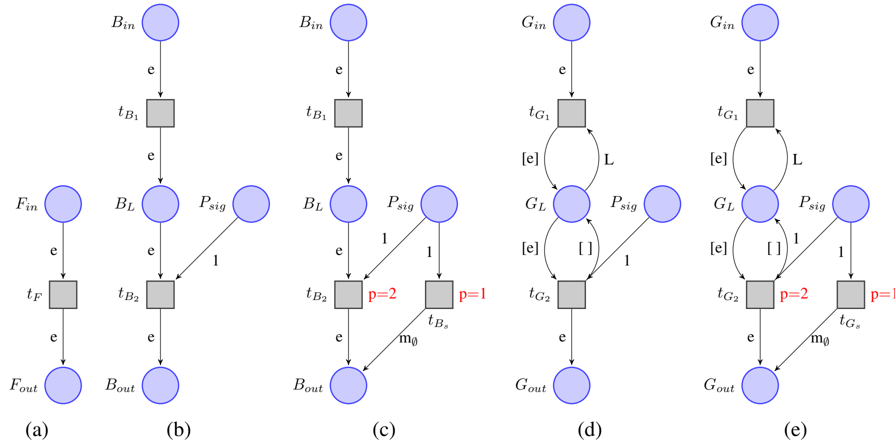


FIG. 5 – *sCFN de sFIFO (a), bBuffer (b), nbBuffer (c), bGreedy (d) and nbGreedy (e).*

Pour les connecteurs de type *Greedy* et *Buffer* le $sCFN$ doit modéliser soit la perte soit la bufferisation des messages. Ainsi ils sont modélisés à partir de deux transitions pour respectivement consommer les données qui arrivent et extraire le message à envoyer en sortie. La disponibilité d'une donnée sur le port d'entrée d'un connecteur est modélisé par l'existence d'un jeton dans la place B_{in} (resp. G_{in}). Ce jeton va être transporté par la transition t_{B_1} (resp. t_{G_1}) vers la place B_L (resp. G_L) qui va stocker le nouveau jeton et le restituer plus tard dans le bon ordre. Par contre G_L qui contient un jeton de type liste, ne stocke que le dernier jeton reçu en écrasant l'ancien. La délivrance d'une donnée au niveau du port de sortie de données

se réalise s'il existe un jeton dans la place P_{sig}. Cette place correspond au port s des connecteurs de type *Buffer* ou *Greedy*. La place P_{sig} avec $C(P_{sig}) = \{Int\}$ représente le port s des connecteurs et la présence d'un jeton dans P_{sig} ainsi que la présence d'un jeton dans la place B_L (resp. un jeton différent de la liste vide dans la place G_L) déclenchent la transmission d'un jeton vers la place B_{out} (resp. G_{out}).

Pour les connecteurs en mode *bloquant*, l'absence de jetons dans la place B_L (resp. une liste vide dans la place G_L) ou dans la place P_{sig} rend la transition t_{B_2} (resp. t_{G_2}) non activable comme illustré dans la figure 5(b) (resp. 5(d)). Pour les connecteurs en mode *non bloquant*, la place P_{sig} est également liée à la place B_{out} (resp. G_{out}) pour permettre au connecteur de générer des messages vides $m_\emptyset$ grâce à la transition t_{B_s} (resp. t_{G_s}), comme illustré dans la figure 5(c) (resp. 5(e)). Mais si les deux places B_L (resp. G_L) et P_{sig} sont marquées, les deux transitions t_{B_2} (resp. t_{G_2}) et t_{B_s} (resp. t_{G_s}) sont alors activables. La sémantique de $sCFN$ va déterminer que seule la transition t_{B_2} (resp. t_{G_2}) est déclenchée car son poids est 2 ce qui la rend prioritaire par rapport à t_{B_s} (resp. t_{G_s}).

3.4 Modèle d'une application en sCFN

Il s'agit maintenant de décrire comment se modélise un graphe d'application en $sCFN$. Pour cela nous allons définir la fonction $sCFN(G)$ où G est un graphe d'application défini par $(Comp \bigcup Conn, Dl \bigcup Tl)$. Il s'agit principalement de modéliser les liens de $Dl \bigcup Tl$. Cette opération s'effectue en trois étapes :

1. la transformation des composants et des connecteurs en $sCFN$
2. l'ajout aux composants des transitions permettant la duplication des messages en sortie et l'ajout de messages en entrée pour la synchronisation,
3. la connexion proprement dite des liens de $Dl \bigcup Tl$ sur ces transitions.

Avant toute chose, nous allons considérer que tous les composants et tous les connecteurs ont des noms de port différents (pour cela il suffit de considérer par exemple que chaque composant et chaque connecteur a un identifiant unique et que le nom de chacun des ports de ces objets est préfixé par cet identifiant). Ainsi dans la suite, pour simplifier les notations, les liens sont représentés par le couple $\langle p, q\rangle$ au lieu du couple $\langle x^p, y^q\rangle$ (voir définition 5).

L'union de deux $sCFN$ $sCFN_1$ et $sCFN_2$ ayant des places et des transitions disjointes, noté $sCFN_1 \cup sCFN_2$, est le $sCFN$ dont chaque composante est l'union des composantes correspondantes de $sCFN_1$ et $sCFN_2$. L'ensemble P est l'union des places de $sCFN_1$ et $sCFN_2$, l'ensemble T est l'union des transitions de $sCFN_1$ et $sCFN_2$ et ainsi de suite. Pour un ensemble de composants et de connecteurs $\mathcal{C}$, le réseau $sCFN_\cup(\mathcal{C})$ dénote le $sCFN$ défini par $\bigcup_{C\in\mathcal{C}} sCFN(C)$. La première étape de l'algorithme consiste donc simplement à calculer $sCFN_\cup(Comp \bigcup Conn)$.

Considérons un composant C de $Comp$, on note $\overline{pIn}_C$ l'ensemble des ports de données d'entrée connectés de C dans G, c-à-d. $\overline{pIn}_C = \{p_{in} \in pIn_C | \exists p_{out}, \langle p_{out}, p_{in}\rangle \in Dl\}$. L'ensemble de tous les ports d'entrée connectés de l'application est noté $\overline{pIn}(G)$ et se définit par $\overline{pIn}(G) = \bigcup_{c\in Comp} \overline{pIn}_C$. Ces ensembles vont nous permettre de définir les transitions utiles à la deuxième étape de la transformation.

Les liens de données vont être partitionnés pour séparer les liens aboutissant au port *i* d'un connecteur et les liens aboutissant à un port d'entrée d'un composant de la manière suivante :

Entrée : Un graphe d'application $(Comp \bigcup Conn, Dl \bigcup Tl)$
Sortie : $sCFN\ (P, T, A, \Sigma, V, C, E, I)$

Let $sCFN_{\cup}(Comp \bigcup Conn) = (P, T, A, \Sigma, V, C, E, I)$;
Pour chaque $c \in Conn$ **Faire**
  Si $Type(c) \neq sFIFO$ **Alors**
    $T := T \cup \{t_s, t_o\}$;
    $AddArc(\langle t_s, p_s \rangle, 1, A, E)$;
    $AddArc(\langle p_o, t_o \rangle, 1, A, E)$;
    // où s est le port de déclenchement de c

Pour chaque $C \in Comp$ **Faire**
  // Pour chaque port d'entrée connecté de C, ajout de la transition permettant d'ajouter un msg dans la place
  Pour chaque $p \in \overline{pIn_C}$ **Faire**
    $T := T \cup \{t_p\}$;
    $AddArc(\langle P_p, t_p \rangle, \langle x_p, m_p \rangle, A, E)$;
  // Pour chaque port de sortie de C, ajout de la transition pour diffuser le msg produit
  Pour chaque $p \in pOut_C$ **Faire**
    $T := T \cup \{t_p\}$;
    $AddArc(\langle t_p, P_p \rangle, \langle x_p, m_p \rangle, A, E)$;

// Étape 3
Pour chaque $\langle e, s \rangle \in Tl$ **Faire**
  // Création d'une place pour accueillir le signal émis par e
  $P := P \cup \{P_{es}\}$;
  // Duplication du signal émis par e vers l'extérieur
  $AddArc(\langle t_{es}, P_{\langle e,s \rangle} \rangle, 1, A, E)$;
  // Transmission du signal vers la transition de synchronisation du destinataire
  $AddArc(\langle P_{\langle e,s \rangle}, t_s \rangle, 1, A, E)$;

// Envoi des msg vers les ports d'entrée des composants
Pour chaque $\langle s, d \rangle \in InComp(Dl)$ **Faire**
  $AddArc(\langle p_s, t_d \rangle, \langle x_d, m_d \rangle, A, E)$;

// Envoi des msg vers les ports d'entrée des connecteurs
Pour chaque $\langle s, d \rangle \in InConn(Dl)$ **Faire**
  $AddArc(\langle t_s, P_d \rangle, \langle x_s, m_s \rangle, A, E)$;

Algorithme 2 : Transformation d'un graphe d'application en $sCFN$

$InConn(Dl) = \{\langle p_1, p_2 \rangle \in Dl \mid \exists c \in Conn, p_2 \in pIn(c)\}$
$InComp(Dl) = \{\langle p_1, p_2 \rangle \in Dl \mid \exists C \in Comp, p_2 \in pIn(C)\}$.
De même, on va distinguer les liens déclencheurs aboutissant sur le port *s* d'un composant des

autres. On notera ces deux ensembles $InConn(Tl)$ et $InComp(Tl)$.

Ces ensembles vont nous être utiles pour définir la troisième étape de la transformation et distinguer les différents types de liens. Les trois étapes de la transformation $sCFN(G)$ où G est un graphe d'application est finalement donné sous forme d'un algorithme (Algorithme 2).

Pour montrer que $sCFN(App)$ modélise bien la sémantique définie dans la section 2.4, il suffit d'observer le comportement des réseaux représentant les liens entre les éléments de l'application. Lorsqu'un message est disponible sur un port p de données en sortie d'un composant ou d'un connecteur, alors la transition t_p correspondante est activable, donc le message pourra franchir la transition et aller se placer dans toutes les places de sortie de t_p. Lorsqu'un composant émet un signal sur son port e, ce signal sera dupliqué dans toutes les places P_{es} par la transition t_s du composant, puis ce signal sera acheminé dans la place P_s du composant (ou dans la place P_{sig} du connecteur) grâce à la transition t_s correspondante. Cette dernière transition ne sera activable que si un signal se trouve dans chacune de ses places d'entrée ce qui effectuera la synchronisation. Par conséquent, étant donné que $sCFN(C)$ modélise la sémantique de C pour tout $C \in Comp \bigcup Conn$ et que les $sCFN$ modélisant les liens $sCFN(App)$ sont conformes à la sémantique des liens dans App, le réseau FIFO coloré strict $sCFN(App)$ modélise bien le comportement de l'application App.

4 Conclusion

Dans cet article, nous avons défini un modèle de composants itératifs permettant de spécifier des applications de visualisation scientifique interactives. Nous avons aussi défini une classe particulière des réseaux FIFO colorés appelée réseaux FIFO colorés stricts ($sCFN$). Nous avons donné un algorithme permettant de décrire la sémantique opérationnelle d'une application définie dans notre modèle par un $sCFN$. Ce travail est une première étape qui nous permet d'avoir une modélisation formelle des applications de visualisation scientifique interactives qui repose sur des outils solides dans le domaine de la vérification. Il nous permet déjà d'avoir un premier simulateur de nos applications en utilisant un logiciel comme CPNTools de Ratzer et al. (2003) ou TINA de Berthomieu et al. (2004).

Dans un premier temps, nous aimerions enrichir notre modèle de composants afin de pouvoir prendre en compte des composants parallèles et des composants hiérarchiques très largement utilisés en particulier pour la partie modélisation pour avoir des applications performantes et interactives suivant différents niveaux d'abstraction.

Mais notre ambition générale est de construire une boîte à outils permettant d'assister les concepteurs d'applications de visualisation scientifique interactive dans la construction de leur application. D'un côté, l'utilisateur spécifie ces applications à partir du graphe d'application qui est intuitif, et de l'autre, nous lui fournirons des outils permettant de l'aider à vérifier son bon fonctionnement ou de transformer l'application initiale pour l'optimiser. La validité de ces outils reposera sur la sémantique opérationnelle en $sCFN$ de l'application. Par exemple, l'initialisation d'une application n'est pas toujours simple car elle contient souvent des cycles. En se basant sur le $sCFN$ d'une application et leurs propriétés de vivacité, nous pourrons définir une méthode automatique permettant le démarrage d'une application.

De plus, nous nous intéressons désormais aux applications dynamiques. Par exemple, en analyse visuelle (visual analytics), les applications impliquent la gestion de composants qui apparaissent ou disparaissent en cours d'exécution, pour effectuer une étape ponctuelle d'analyse

des données par exemple. Cette insertion dynamique nécessite de la reconfiguration à la volée pour définir les conditions d'arrivée et de sortie des nouveaux composants. En particulier, il s'agit d'être capable de mettre en pause les composants impactés et de définir les conditions de reprise de l'application modifiée. L'intégrité de l'application doit être garantie pour chacune des différentes étapes de cette reconfiguration. Grâce à la modélisation en $sCFN$, nous pourrons définir des outils qui permettent ses opérations et dont la validité pourra être démontrée.

L'ensemble de ces perspectives repose maintenant sur une étude des $sCFN$, notamment sur les conditions de leur vivacité (le fait que toutes les transitions pourront être appliquées à un moment donné) d'une part et de leur exploitation dans les applications visées d'autre part. De premiers résultats sur la mise en pause d'une application semblent prometteurs dans ce sens.

Références

Allard, J., V. Gouranton, L. Lecointre, S. Limet, B. Raffin, et S. Robert (2004). FlowVR : A middleware for large scale virtual reality applications. In M. Danelutto, M. Vanneschi, et D. Laforenza (Eds.), *Euro-Par*, Volume 3149 of *Lecture Notes in Computer Science*, pp. 497–505. Springer.

Arbab, F. (2004). Reo : a channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14*(3), 329–366.

Armstrong, R., G. Kumfert, L. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, et T. Dahlgren (2006). The CCA component model for high-performance scientific computing. *Concurrency and Computation : Practice and Experience 18*(2), 215–229.

Benalycherif, M.-L. et C. Girault (1996). Behavioural and structural composition rules preserving liveness by synchronization for colored fifo nets. In *Application and Theory of Petri Nets*, pp. 73–92.

Berthomieu, B., P.-O. Ribet, et F. Vernadat (2004). The tool TINA – construction of abstract state spaces for petri nets and time petri nets. *Journal of Production Research 42*(14), 2741–2756.

Blair, G., T. Coupaye, et J.-B. Stefani (2009). Component-based architecture : the fractal initiative. *annals of telecommunications - annales des télécommunications 64*(1-2), 1–4.

Borgdorff, J., M. Mamonski, B. Bosak, D. Groen, M. B. Belgacem, K. Kurowski, et A. G. Hoekstra (2013). Multiscale computing with the multiscale modeling library and runtime environment. *Procedia Computer Science 18*(0), 1097 – 1105.

Dormoy, J., O. Kouchnarenko, et A. Lanoix (2011). Using Temporal Logic for Dynamic Reconfigurations of Components. In *7th International Workshop on Formal Aspects of Component Software - FACS'2010*, Guimaraes, Portugal.

Emmerich, W. et N. Kaveh (2002). Component technologies : Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, New York, pp. 691–692. ACM Press.

Finkel, A. et G. Memmi (1982). Fifo nets : A new model of parallel computation. In A. Cremers et H.-P. Kriegel (Eds.), *Theoretical Computer Science*, Volume 145 of *Lecture Notes in Computer Science*, pp. 111–121. Springer Berlin Heidelberg.

Goodale, T., G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, et J. Shalf (2003). The cactus framework and toolkit : design and applications. In *Proceedings of the 5th international conference on High performance computing for computational science*, VECPAR'02, Berlin, Heidelberg, pp. 197–227. Springer-Verlag.

Hamadi, R. et B. Benatallah (2003). A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference - Volume 17*, ADC '03, Darlinghurst, Australia, Australia, pp. 191–200. Australian Computer Society, Inc.

Heinzemann, C., C. Priesterjahn, et S. Becker (2012). Towards modeling reconfiguration in hierarchical component architectures. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, New York, pp. 23–28. ACM.

Jensen, K. et L. M. Kristensen (2009). *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer.

Léger, M., T. Ledoux, et T. Coupaye (2010). Reliable dynamic reconfigurations in a reflective component model. In L. Grunske, R. Reussner, et F. Plasil (Eds.), *Component-Based Software Engineering*, Volume 6092 of *Lecture Notes in Computer Science*, pp. 74–92. Springer.

Limet, S., S. Robert, et A. Turki (2011). Controlling an iteration-wise coherence in dataflow. In F. Arbab et P. C. Ölveczky (Eds.), *FACS*, Volume 7253 of *Lecture Notes in Computer Science*, pp. 241–258. Springer.

Ludascher, B., I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, et Y. Zhao (2006). Scientific workflow management and the Kepler system. *Concurrency and Computation : Practice and Experience 18*(10), 1039–1065.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA : Prentice Hall PTR.

Ratzer, A. V., L. Wells, H. M. Lassen, M. Laursen, J. Frank, M. S. Stissing, M. Westergaard, S. Christensen, et K. Jensen (2003). CPN tools for editing, simulating, and analysing coloured petri nets. In *Applications and Theory of Petri Nets 2003 : 24th International Conference, ICATPN 2003*, pp. 450–462. Springer Verlag.

Summary

The component-based approaches are now widespread in software engineering. In the context of interactive scientific visualisation applications, these approaches are well-suited because of the separation of concerns that they offer. The modeling of such applications need to describe the component behaviours and the way they cooperate to make an application. This cooperation is expressed as a communication network which can be very complex with specific policies that may lose messages to improve application performance. We propose a dedicated component approach for scientific visualisation applications associated to a formalisation based on colored FIFO nets. The model aims to describe the component behaviors and the communication network and the formalisation aims to offer verification tools.