Construction automatique d'applications de visualisation scientifique interactive fortement cohérentes

Sébastien Limet*, Sophie Robert*, Ahmed Turki* [1]

*Laboratoire d'Informatique Fondamentale d'Orléans,
Batiment IIIA, Rue Léonard de Vinci, 45067 ORLEANS
{sebastien.limet | sophie.robert | ahmed.turki}@univ-orleans.fr

Résumé. Cet article traite de l'usage de la programmation par composants pour la conception d'applications de visualisation scientifique interactive temps-réel. Notre but est d'automatiser cette conception grâce à une méthode pour construire un réseau de connexion entre les différents composants qui constituent une telle application à partir des contraintes fournies par l'utilisateur. Ce type d'applications doit fonctionner le plus rapidement possible tout en préservant la précision de ses résultats. Ces deux aspects sont souvent en conflit, par exemple lorsqu'il s'agit d'autoriser la perte de messages ou pas. Notre approche vise à trouver automatiquement le meilleur compromis entre ces deux critères au moment de construire l'application.

1 Introduction

La visualisation interactive de simulations aide les scientifiques à mieux comprendre les phénomènes qu'ils étudient. Par exemple, les biochimistes Delalande et al. (2010) décrivent comment elle peut révéler des interactions entre des structures moléculaires complexes. L'observateur, en devenant acteur de l'application, peut non seulement naviguer dans la visualisation mais également appliquer des forces sur les atomes, forces qui agissent directement sur la simulation en cours. Il peut ainsi pousser ou tirer des atomes et constater directement l'implication de ses actions au niveau moléculaire. D'autres interfaces (sonores, haptiques, etc.) peuvent compléter le dispositif. Comme l'a montré McAdam (2010), la force et le retour de force peuvent ainsi être utiles pour comprendre les lois qui régissent un système dynamique.

C'est la raison pour laquelle permettre aux scientifiques de développer leurs simulations puis de les intégrer dans des applications de visualisation interactive est de plus en plus important. Cependant, la complexité de la mise en œuvre de telles applications est un vrai frein à leur développement. Les spécialistes du calcul scientifique sont rarement experts à la fois en rendu graphique, en interaction et en couplage de codes. La programmation par composants -ou *orientée composant* (POC)- est une solution puisqu'elle permet à chaque spécialiste de développer indépendamment les fonctionnalités pour lesquelles il est compétent. L'application *composite* (ou *modulaire*) est alors constituée d'un ensemble de *composants* ou *modules* indépendants mais qui communiquent entre eux selon un *protocole*. Le choix d'une architecture

1. Ce travail est financé par le projet ANR "FvNano".

par composants pour ce genre d'applications est également dû à l'éventualité de lancer ces différentes tâches de manière distribuée. Les composants les plus gourmands en ressources tels que la simulation et l'affichage peuvent ainsi bénéficier de ressources matérielles spécifiques.

L'approche par composants est largement utilisée dans le monde du calcul scientifique pour concevoir des applications composites. Celles-ci suivent très souvent un modèle de flux de travaux (*workflow*, en anglais) (Bouziane et al. (2008)). Ce modèle assimile l'application à une chaîne de traitements dans laquelle des données sont passées d'une tâche à une autre lorsqu'une condition est remplie -par exemple lorsque la tâche courante est accomplie. Lorsque le déclenchement d'une tâche est conditionné par la réception d'une donnée, on parle d'un flux de travaux *flot de données*. Dans cet article, nous souhaitons formaliser un modèle de flux de travaux pour applications de visualisation scientifique interactive. Il vise à faciliter l'intégration de codes hétérogènes. Dans ce type d'applications, ces codes sont souvent itératifs :

- La simulation d'une expérience scientifique consiste en une mise à jour d'un environnement physique au fur et à mesure que ses paramètres internes évoluent ou que des facteurs externes y sont injectés ;
- Le programme d'affichage met, à fréquence constante, l'image affichée à jour, qu'elle ait changé ou pas ;
- Le module d'interaction est une boucle qui vérifie, à fréquence régulière, l'arrivée de nouvelles actions utilisateur.

Ainsi il est important que le composant du modèle tienne compte de ce fonctionnement itératif des codes dans sa définition.

Le modèle doit également favoriser la performance de l'application. Les composants d'une application sont hétérogènes et peuvent fonctionner à des fréquences très différentes. Par exemple, les simulations sont souvent relativement lentes tandis que les composants d'interfaçage avec un périphérique itèrent très vite. Différents schémas de communication doivent permettre divers degrés de synchronisation entre composants. La performance des applications ne doit cependant pas occulter la fiabilité attendue des résultats d'une application scientifique. L'un des moyens d'obtenir de la performance dans un flux de travaux est d'autoriser la perte de données par certaines connexions. Toutefois, une perte non contrôlée de données peut rapidement altérer la cohérence des résultats affichés. La notion de cohérence que nous entendons ici se définit comme la synchronisation de différents flots de données entre deux composants. Ce type de préoccupations en rapport avec la provenance des données est bien connu des concepteurs de flux de travaux scientifiques dont Yildiz et al. (2009). Il apparaît donc nécessaire que le modèle prévoie des mécanismes permettant de surveiller, contrôler voire supprimer la perte de données au niveau de n'importe quelle connexion afin de satisfaire une contrainte de cohérence imposée par l'utilisateur.

Ajoutons, enfin, que le travail de composition doit rester accessible pour l'utilisateur, même lorsque l'application à construire est complexe. Par exemple, il est possible de réfléchir à un modèle de composition -le modèle accessible à l'utilisateur- proposant un certain degré d'abstraction par rapport au modèle de composants complet.

De nombreux systèmes de gestion de flux de travaux (*Scientific Workflow Management Systems* ou *SWfMS*, en anglais) (Deelman et al. (2009)) sont proposés pour concevoir, générer voire déployer et exécuter des applications scientifiques. Par exemple, SCIRun par Weinstein et al. (2005) et CoViSE par Wierse (1996) se définissent comme des environnements pour la simulation et la visualisation de problèmes scientifiques. Dans ces environnements, l'exécution

de l'application demeure cependant résolument séquentielle et synchrone. De plus, l'interaction correspond à du pilotage (Mulder et al. (1999)) direct des modules de simulation. D'autres modèles tels que ceux de Triana par Churches et al. (2006) et de Kepler par Ludascher et al. (2006) offrent plus de possibilités en matière de coordination. Le premier ne définit pas de comportement standard pour son composant. L'utilisateur a donc la liberté de programmer les composants de son application en fonction du déroulement global qu'il souhaite. Ce choix de conception nuit cependant à la réutilisabilité des composants. Avec Kepler, cette liberté est plus encadrée puisque l'utilisateur ne peut que choisir la même politique de synchronisation pour toutes les connexions de l'application. Malgré cela, Triana et Kepler restent fondamentalement des systèmes de traitement par lot (*batch processing*, en anglais) et non de visualisation ou de pilotage de simulations.

Pour être en mesure d'intégrer des simulations continues ou des périphériques d'interaction de manière désynchronisée, un système flot de données doit savoir gérer des flux de données continus émis par ses composants. On peut ici prendre pour exemple l'architecture logicielle du système FlowVR pour la réalité virtuelle par Allard et al. (2010). Quelques travaux essaient d'ajouter cette capacité aux systèmes destinés au calcul scientifique, par exemple par l'extension d'un paradigme flux de données existant tel qu'ont fait Barseghian et al. (2010). Dans la plupart de ces travaux, ces développements sont motivés par la nécessité d'alimenter des flux de travaux en données issues de capteurs (e.g. GPS). Nous pensons que le besoin est le même pour créer des flux de travaux de visualisation scientifique interactive eu égard aux caractéristiques des composants citées plus haut.

Dans cet article, nous présentons un système pour spécifier des applications composites de visualisation scientifique interactive. Il est organisé de la manière suivante : la section 2 présente les éléments de notre modèle et les différents graphes décrivant l'application. Dans la section 3, nous formalisons notre notion de cohérence et expliquons les étapes de sa mise en œuvre automatique. La section 4 donne quelques résultats expérimentaux de notre modèle et de notre méthode appliqués à un cas réel. Enfin, dans la section 5, nous évaluons notre approche et donnons quelques uns de nos futurs axes de recherche.

2 Éléments du modèle

Notre modèle de composants comporte un élément de traitement appelé *composant*, plusieurs objets de communication appelés *connecteurs* ainsi que des *liens* pour relier composants et connecteurs. La structure et le comportement de ces éléments ont été pensés pour remplir les objectifs de performance et de cohérence scientifique inhérentes aux applications de visualisation scientifique interactive. Une application est représentée par un assemblage de ces éléments appelé *graphe d'application*. Après avoir présenté le modèle opérationnel, nous en donnerons une abstraction qui permettra au programmeur de spécifier, à travers un *graphe de spécification*, la politique de communication qu'il souhaite mettre en œuvre au sein de l'application.

2.1 Les composants

Un composant encapsule une tâche dans la/les chaîne(s) de traitement de l'application et est itératif par nature. Formellement, le composant se définit comme un quadruplet $C = (n, \{I \cup \{s\}\}, \{O \cup \{e\}\}, f)$ avec :

- n son identifiant unique ;
- I l'ensemble de ses ports d'entrée
- s un port de déclenchement ;
- O l'ensemble de ses ports de sortie ;
- e un port de signalement ;
- f un booléen qui prend la valeur "vrai" si le composant est interactif. Cet attribut doit être activé si le composant gère des périphériques d'entrée. Dans ce cas, le composant n'itère que pendant que l'utilisateur manipule le périphérique. Il ne doit alors pas être ralenti par un autre composant ou par un connecteur. De même, il ne doit pas non plus être saturé de messages lorsqu'il est inactif.

Les composants reçoivent et émettent des données sous la forme de *messages*. Comme l'illustre la figure 1, l'itération du composant se déroule de la manière suivante :

1. Attendre d'avoir reçu un nouveau message sur chacun de ses ports d'entrée de données connectés ;
2. Attendre sur son port s, s'il est connecté, un signal de déclenchement ;
3. Récupérer les données reçues en entrée et exécuter sa/ses tâche(s) ;
4. Envoyer des données-résultats à travers tous ses ports de sortie ainsi que, via le port e, un signal notifiant la fin d'une itération.

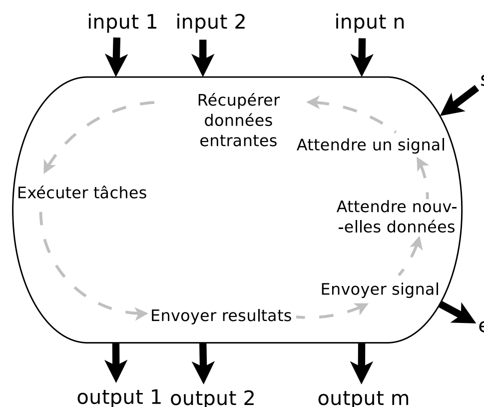

FIG. 1: Le cycle d'itération du composant.

Les ports de données d'un composant portent également un identifiant. En plus des données à transmettre, un message peut contenir des informations annexes appelées estampilles. Ces estampilles peuvent être propres à l'application et définies par l'utilisateur dans l'implémentation des composants. Notre modèle prévoit, cependant, une estampille par défaut qui est le numéro $it(m)$ de l'itération du composant lorsqu'il a émis le message m. Enfin, le composant accepte aussi les messages vides -sans données- auquel cas il itère et son comportement est défini par l'utilisateur à l'implémentation.

Dans la suite de ce papier, nous emploierons les notations suivantes : $I(C)$ et $O(C)$ désignent, respectivement, les ensembles de ports d'entrée et de sortie d'un composant C tandis que $C.p \in I(C) \cup O(C)$ représente le port p de C.

2.2 Les connecteurs

Un connecteur doit être placé entre deux composants qu'on souhaite faire communiquer et détermine le niveau de synchronisation entre eux. Formellement, un connecteur est un quadruplet $c = (n, \{i, s\}, \{o\}, t)$ avec :
- n son identifiant unique ;
- i son unique port d'entrée de données ;
- s son port de déclenchement ;
- o son unique port de sortie de données ;
- t son type.

Nous emploierons pour le connecteur, lorsqu'elles s'appliquent, les mêmes notations que pour le composant. En outre, $type(c)$ désignera le type d'un connecteur c.

Nous avons choisi pour notre modèle cinq types de connecteurs afin de couvrir les cas d'utilisation les plus courants. D'abord, les connecteurs doivent pouvoir éviter la saturation de messages en entrée du composant récepteur lorsque le composant émetteur est plus rapide que lui. Pour le prémunir de cela, trois solutions classiques existent dans la littérature, notamment chez Arbab et al. (2007) et Pautasso et Alonso (2006) :

1. Autoriser le connecteur à jeter des messages tant que le récepteur n'est pas prêt à en accepter ;
2. Contraindre l'émetteur à vérifier la disponibilité du récepteur avant d'itérer ;
3. Équiper le connecteur d'un buffer capable de stocker un certain nombre de messages en attente de traitement.

En optant pour la première ou la deuxième de ces solutions, l'utilisateur choisira, au profit de la stabilité de l'application, de sacrifier soit l'intégrité des données transmises par le connecteur soit la performance de l'émetteur. Par ailleurs, bien qu'elle soit la plus sûre, la deuxième option, si elle est généralisée dans l'application, est susceptible de limiter la vitesse de tous les composants à celle du plus lent comme le font remarquer Pautasso et Alonso (2006). La troisième option, elle, n'est à prescrire que si l'on sait que l'écart de vitesse peut s'inverser et laisser la possibilité au récepteur de consommer le contenu du buffer.

Un modèle destiné aux applications de visualisation interactive doit aussi permettre à un composant de fonctionner à sa propre fréquence même si les composants qui l'alimentent en messages sont plus lents. Une solution peut être mise en place sans aller à l'encontre de la logique flot de données. Elle consiste à charger le connecteur de fournir des messages vides à ce composant si jamais ce dernier est prêt et que l'émetteur, lui, n'a pas encore envoyé de nouveaux messages. Typiquement, un tel connecteur peut être placé en aval d'un composant d'interaction qui n'émet de messages que suite à une action de l'utilisateur.

Nos connecteurs, rassemblés dans la figure 2, sont les suivants :
- Le sFIFO est une connexion FIFO dans laquelle l'émetteur attend sur son port s un signal de déclenchement avant d'itérer et ceci pour éviter toute saturation. Ce connecteur impose une synchronicité parfaite entre émetteur et receveur ;
- Le bBuffer est une pile FIFO qui empile tous les messages arrivant et n'en délivre un que s'il a reçu un signal de déclenchement sur son port s. Ce schéma de communication peut-être utile lorsqu'au moins un des deux composant itère de manière irrégulière ;

– Le nbBuffer est similaire au bBuffer et a, en plus, la capacité de générer un message vide pour le récepteur lorsqu'il est déclenché à vide. Il émet également un message vide à sa toute première itération ;
– Le bGreedy ne stocke que le dernier message reçu et, comme le Buffer, le délivre sur ordre du composant receveur. Un tel connecteur sert à empêcher la saturation du receveur lorsqu'il n'est pas nécessaire que tous les messages soient transmis ;
– Le nbGreedy est la variante non-bloquante du Greedy.

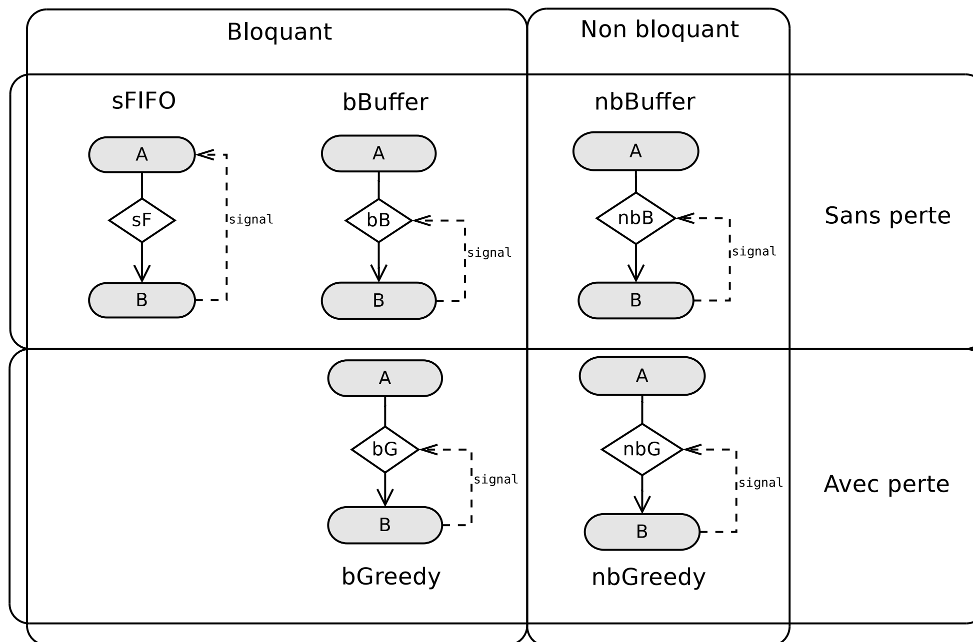

FIG. 2: Les cinq connecteurs.

2.3 Les liens

Les liens relient composants et connecteurs entre eux via leurs ports. Ils sont notés $(x.o, y.i)$ avec x et y des composants ou connecteurs et $o \in O(x)$ et $i \in I(y)$. Il existe deux types de liens :

Les liens de données véhiculent des messages. Pour un lien de données $(x.o, y.i)$, nous imposons que $o \neq e$, $q \neq s$ et qu'au moins x ou y ne soit pas un composant. En effet, un connecteur est toujours requis pour définir la politique de communication entre deux composants. Un port d'entrée ou de sortie d'un connecteur n'est connecté qu'à un seul lien de données.

Les liens de déclenchement véhiculent des signaux de déclenchement. Ils concrétisent le flot de contrôle dans notre modèle. Pour un tel lien $(x.e, y.s)$, nous imposons que x soit un

composant. Par ailleurs, si un port s est connecté à plusieurs liens de déclenchement, il faudra qu'un signal soit reçu par chacun de ces liens pour que le déclenchement soit effectif. De plus, pour éviter les inter-bloquages, composants et connecteurs n'attendent pas de signal de déclenchement pour effectuer leur toute première itération.

2.4 Graphe d'application

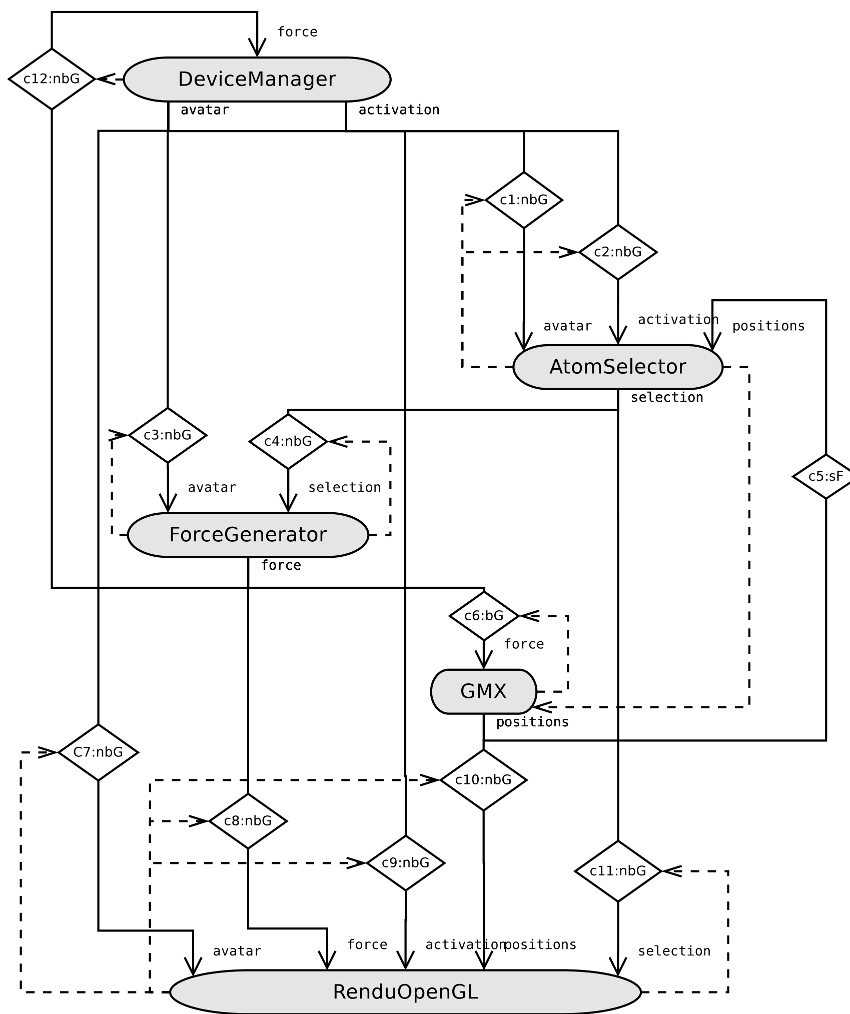

FIG. 3: Un graphe d'application.

Une application est représentée par un graphe d'application à partir des éléments précédents. Ses sommets sont les composants et les connecteurs. Ses arêtes sont les liens. Formelle-

ment, $\mathcal{G} = (\mathcal{C} \cup \mathcal{L}, \mathcal{D} \cup \mathcal{T})$ définit un graphe d'application avec $\mathcal{C}$ un ensemble de composants, $\mathcal{L}$ un ensemble de connecteurs, $\mathcal{D}$ un ensemble de liens de données et $\mathcal{T}$ un ensemble de liens de déclenchement.

La figure 3 montre un graphe d'application possible pour une application de visualisation scientifique interactive dans lequel nous utilisons différents types de connecteurs. Il est constitué de :

– une simulation **GMX** ;
– un composant de visualisation **RenduOpenGL** qui ne propose qu'une interaction de type navigation (clavier/souris). Ce module ne gère pas directement les autres interactions mais va en rendre compte grâce aux informations qu'il reçoit sur ses ports d'entrée ;
– un composant interactif **DeviceManager** relié à des périphériques (un bras haptique, par exemple) qui envoie aux composants auxquels il est relié la position de l'avatar ainsi que la liste des boutons activés ;
– un composant **AtomSelector** permettant de sélectionner des atomes en fonction de la position des atomes et de l'avatar et de la pression sur un bouton ;
– un composant **ForceGenerator** actualisant la direction et l'intensité des forces à appliquer à la simulation.

Pour appliquer des forces à la simulation, l'utilisateur doit bouger le périphérique en maintenant son bouton enfoncé. Le composant d'interaction émet alors des données continuellement et rapidement. Nous utilisons des nbGreedy en sortie de ce composant afin de ne pas saturer les composants qu'il alimente mais aussi pour leur permettre de fonctionner lorsqu'il est au repos. La visualisation est également désynchronisée du reste de l'application au moyen de nbGreedy afin d'être la plus fluide possible. Ensuite, un bGreedy relie le générateur de forces au composant de calcul car il est plus rapide que lui. Les forces sont également renvoyées au composant d'interaction afin de produire un retour de force au niveau du périphérique.

Pour la suite de cet article, nous considérerons également les définitions suivantes :

Définition 1 (Chemin de données) *Un chemin de données dans le graphe d'application $\mathcal{G}$ est un chemin acyclique dans le graphe $(\mathcal{C} \cup \mathcal{L}, \mathcal{D}) \subset \mathcal{G}$.*

Le chemin $P = (DeviceManager, c_3, ForceGenerator, c_8, RenduOpenGL)$ est un exemple de chemin de données sur la figure 3. Nous définissons également la source et la destination d'un chemin de données.

Définition 2 (Source et destination d'un chemin de données) *La source $src(P)$ est le sommet de départ d'un chemin de données P dans $\mathcal{G}$ et la destination $dest(P)$ son sommet d'arrivée.*

Les composants **DeviceManager** et **ForceGenerator** sont, respectivement, la source et la destination du chemin P sur la figure 3. Nous définissons, enfin, le message résultat d'un chemin et la position d'un sommet dans un chemin.

Définition 3 (Résultat d'un chemin de données) *Un message m arrivant à $dest(P)$ est appelé résultat de P et le message issu de $src(P)$ et à l'origine de ce résultat est noté $ori_P(m)$.*

2.5 Graphe de spécification

La composition d'une application selon notre modèle de composants peut se faire par instanciation et liaison des différents éléments définis à la section 2. Toutefois, nous proposons

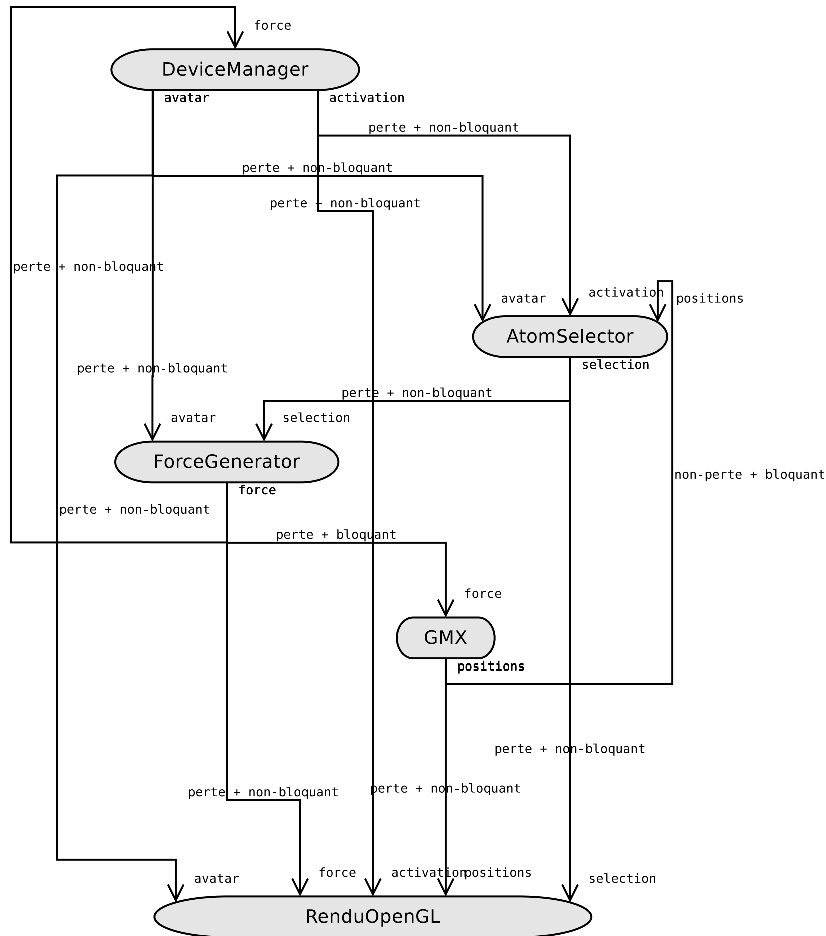

FIG. 4: Un graphe de spécification.

une méthode de composition épargnant à l'utilisateur la connaissance de ces éléments et ce, par la réalisation d'un graphe d'un niveau d'abstraction supérieur à celui du graphe d'application appelé le *graphe de spécification*. La spécification d'applications par ce biais permet à l'utilisateur d'exprimer les propriétés qu'il souhaite attribuer à chaque connexion, lui occultant au maximum les technicités du modèle de composants.

Les sommets du graphe de spécification sont les composants du graphe d'application. Les arêtes du graphe, orientées du composants émetteur vers le composant récepteur, sont étiquetées avec les noms des ports de sortie et d'entrée qu'elles relient ainsi qu'avec deux propriétés relatives à la communication :

- La politique en matière de perte de messages : spécifie si la connexion est autorisée à perdre des messages pour prévenir la saturation du composant récepteur ;
- Le comportement bloquant : spécifie si la connexion est autorisée à introduire des mes-

sages vides pour débloquer le composant récepteur en l'absence de nouveaux messages en provenance de l'émetteur.

À partir des informations fournies dans le graphe de spécification, il est possible d'en déduire un graphe d'application. Cela est réalisé en remplaçant chaque arête par le connecteur correspondant à la combinaison des deux propriétés de la connexion et de la valeur des attributs f des composants émetteur et récepteur. Ces remplacements sont dictés par les règles du tableau 1. Le tableau recense, pour chaque combinaison de propriétés, les connecteurs qui la respectent. Dans les cas, majoritaires, où plusieurs connecteurs conviennent à une combinaison, un choix est imposé pour satisfaire le principe suivant : *L'application générée doit, d'abord, prévenir les saturations et, ensuite, être la plus performante possible*. Les connecteurs choisis répondent à ce principe et sont soulignés dans le tableau. Notons également que, comme justifié à la section 2.1, un connecteur non bloquant et avec perte est indispensable en amont d'un composant interactif.

<table>
<tr><th></th><th colspan="2">Bloquant</th><th colspan="2">Non bloquant</th></tr>
<tr><th>Récepteur</th><td colspan="2"></td><th>Interactif</th><th>Non interactif</th></tr>
<tr><th>Perte</th><td colspan="2"><u>bGreedy</u>, bBuffer, nbGreedy, nbBuffer</td><td>nbGreedy</td><td><u>nbGreedy</u>, nbBuffer</td></tr>
<tr><th>Émetteur</th><th>Interactif</th><th>Non interactif</th><td colspan="2"></td></tr>
<tr><th>Sans perte</th><td><u>bBuffer</u>, nbBuffer</td><td><u>sFIFO</u>, bBuffer, nbBuffer</td><td></td><td>nbBuffer</td></tr>
</table>

TAB. 1: Règles de sélection automatique des connecteurs.

La figure 4 illustre un graphe de spécification pour le graphe d'application de la figure 3.

Pour être complet le graphe de spécification pourra intégrer des contraintes de cohérences que nous allons introduire dans la section suivante.

3 Composition avec contraintes de cohérence

Outre les propriétés sur les composants et les connexions, le graphe de spécification permet à l'utilisateur d'imposer des contraintes de cohérence précises sur des parties de l'application. Nous allons désormais décrire comment le graphe d'application construit précédemment est transformé automatiquement en un graphe d'application final qui respecte les contraintes de cohérences de l'utilisateur tout en préservant le plus possible des connecteurs avec perte pour limiter la dégradation de la performance. Une synthèse des étapes de notre construction à partir du graphe de spécification est illustrée Figure 5.

Les contraintes de cohérence sont basées sur la provenance spatiale et temporelle des flots de messages atteignant un même composant désigné par l'utilisateur. Plus précisément, une contrainte de cohérence est le fait d'imposer l'égalité des numéros d'itérations des couples de messages m_1 et m_2 à l'origine de tous les messages arrivant simultanément à deux ports d'entrée d'un même composant. Formellement, la cohérence est définie ainsi (on rappelle que $it(m)$ désigne le numéro d'itération d'un message) :

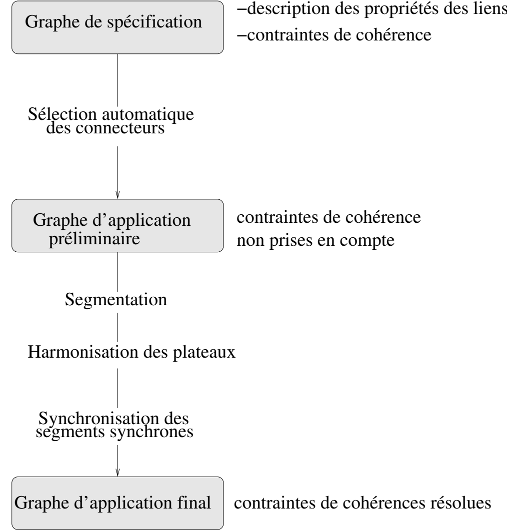

FIG. 5: Les étapes vers un graphe d'application sous contraintes de cohérence.

Définition 4 (Cohérence) *Soient A, B et C trois composants tels que $A.o_i \in O(A), B.o_j \in O(B), C.i_1 \in I(C)$ et $C.i_2 \in I(C)$. La contrainte de cohérence κ définie par*

$$A.o_{i_{\rightarrow C.i_1}} = B.o_{j_{\rightarrow C.i_2}}$$

est satisfaite si, pour chaque couple de chemins P_1 et P_2 débutant, respectivement, par $A.o_i$ et $B.o_j$ et atteignant, respectivement, $C.i_1$ et $C.i_2$, $it(ori_{P_1}(m_1)) = it(ori_{P_2}(m_2))$ avec m_1 et m_2 résultats, respectivement, de P_1 et P_2 lus à la même itération de C. $\mathcal{K}_{\mathcal{G}}$ désigne l'ensemble des contraintes de cohérence auxquelles est soumise une application $\mathcal{G}$.

Cette cohérence revient à demander une synchronisation parfaite entre les flots de données reliant $A.o_i$ à $C.i_1$ et $B.o_j$ à $C.i_2$.

La cohérence de données issues de chemins différents, à leur arrivée au même composant, est une propriété recherchée dans les applications de visualisation scientifique interactive. Elle peut être la garante de l'ergonomie dans des applications interactives très désynchronisées. Par exemple, lorsque l'application permet à l'utilisateur d'influencer la simulation au moyen d'un outil virtuel, une cohérence est souhaitable entre la position de l'avatar représentant à l'écran cet outil et l'état affiché de la simulation que cette position a engendré.

3.1 Sous-graphe de cohérence

À partir des ports de sortie et des ports d'entrée impliqués dans une contrainte de cohérence, il est possible de déduire les chemins de données à rendre cohérents.

Définition 5 (Chemins frères) *ϕ_κ désigne l'ensemble des chemins impliqués dans une contrainte de cohérence κ : $A.o_{i_{\rightarrow C.i_1}} = B.o_{j_{\rightarrow C.i_2}}$. Un couple de chemins, $P_1 \in \phi_\kappa$ entre $A.o_i$ et $C.i_1$ et $P_2 \in \phi_\kappa$ entre $B.o_j$ et $C.i_2$, sont dits chemins frères à l'égard de la cohérence κ. $\phi_\kappa(P)$ désigne l'ensemble des chemins frères d'un chemin P à l'égard de la cohérence κ.*

La première phase de la transformation consiste à construire les ensembles ϕ_κ de chemins concernés par chaque cohérence κ. Notons que si pour une contrainte κ, ϕ_κ ne contient pas de chemins frères, la contrainte est considérée comme non valide.

Définition 6 (Validité d'une contrainte de cohérence) *Une contrainte de cohérence*

$$\kappa : A.o_{i \to C.i_1} = B.o_{j \to C.i_2}$$

est valide si il existe au moins deux chemins P_1 et P_2 dans ϕ_κ tels que P_1 relie $A.o_i$ à $C.i_1$ et P_2 relie $B.o_j$ à $C.i_2$.

Les chemins de ϕ_κ servent ensuite à constituer le sous-graphe de la cohérence κ.

Définition 7 (Sous-graphe de cohérence) *Dans un graphe d'application $\mathcal{G}$, le sous-graphe $\mathcal{G}_\kappa \subset \mathcal{G}$ de la cohérence κ est le graphe formé par tous les chemins contenus dans ϕ_κ.*

La figure 6 illustre le sous-graphe de la cohérence

$$\kappa : DM.avatar_{\to RO.avatar} = DM.avatar_{\to RO.selection}$$

associé au graphe d'application de la figure 3 où DM et RO désignent respectivement les composants DeviceManager et RenduOpenGL.

En présence de plusieurs contraintes de cohérence dans la même application, si au moins deux chemins appartenant à deux contraintes de cohérence différentes ont au moins un connecteur commun, les sous-graphes de ces deux cohérences sont fusionnés. Les deux contraintes, dites jointes, feront alors référence à ce même graphe comme étant leur sous-graphe.

3.2 Transformations pour établir une cohérence

Nous décrivons, dans cette section, les étapes permettant d'instaurer une cohérence à l'intérieur d'un sous-graphe.

3.2.1 Segmentation des chemins

La cohérence de flots de données est très dépendante des connecteurs présents dans les chemins impliqués. Nous distinguons, en particulier, les effets des *segments synchrones* et des *jonctions*.

Définition 8 (Segment synchrone) *Un chemin $(C_1, c_1, \ldots, C_{n-1}, c_{n-1}, C_n)$ où C_i ($1 \leq i \leq n$) est un composant et c_i ($1 \leq i \leq n-1$) est soit un connecteur sFIFO soit un connecteur bBuffer est appelé un segment synchrone. Ces connecteurs sont, d'ailleurs, dits connecteurs synchrones.*

Propriété 1 (Segment synchrone) *Soit $S = (C_1, c_1, \ldots, C_{n-1}, c_{n-1}, C_n)$ un segment synchrone et m_n un message produit par C_n. Alors, $it(m_n) = it(ori_S(m_n))$.*

Cette propriété est évidente puisqu'aucun message n'est perdu à l'intérieur d'un segment synchrone et qu'aucun message vide n'est introduit par ses connecteurs. C_n génère autant de messages que C_1.

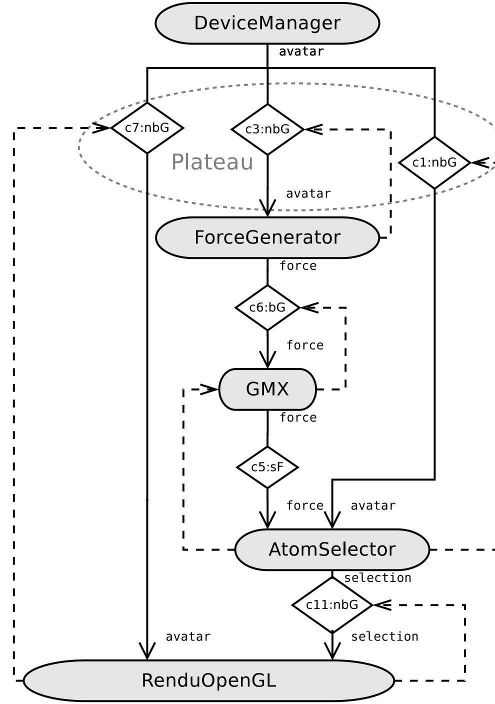

FIG. 6: Sous-graphe de cohérence.

Définition 9 (Jonction et connecteurs synchrones) *Une jonction est un connecteur bGreedy, nbGreedy ou nbBuffer entre deux segments synchrones consécutifs.*

Les jonctions sont, par définition, des points de désynchronisation dans le graphe d'application. C'est à leur niveau que doit donc être contrôlée la cohérence des différents chemins impliqués dans une cohérence. Pour ce faire, une segmentation de ces chemins doit d'abord être établie afin d'obtenir le même nombre de jonctions sur tous ces chemins.

Ce problème de segmentation est résolu par un système d'équations linéaires traitant, en une fois, chaque sous-graphe afin d'éviter au procédé tout retour sur trace entre le traitement d'une cohérence et celui d'une autre.

Dans le système, chaque connecteur c est représenté par une variable v_c. Le domaine de ces variables est $\{0, 1\}$. La valeur 0 signifie que le connecteur est synchrone et la valeur 1 que le connecteur est une jonction. Dès lors, il suffit d'imposer que tous les chemins possibles dans le sous-graphe aient la même somme de variables pour que les chemins frères aient le même nombre de jonctions. Parce que des chemins frères peuvent avoir des portions communes, cette égalisation se fait simultanément entre tous les chemins du sous-graphe et pas seulement entre ceux de ϕ_κ. Cela a pour effet de résoudre le problème globalement.

Formellement, l'ensemble des équations du système linéaire est $Eq_{\mathcal{G}_\kappa} = \bigcup_{\kappa \in \mathcal{K}_{\mathcal{G}_\kappa}} Eq_\kappa$ tel que $Eq_\kappa = \{\sum v_{c_i} = \ldots = \sum v_{c_n} | c_i \in \mathcal{L}_P, P \in \mathcal{G}_\kappa\}$ avec $\mathcal{L}_P$ l'ensemble des connecteurs sur le chemin P.

D'après le tableau 1, les connecteurs précédant un composant récepteur interactif ne peuvent être que de type nbGreedy. Autrement, ce composant risque d'être ralenti par un composant émetteur plus lent ou être saturé par un émetteur plus rapide. Les variables de ces connecteurs sont, en conséquence, fixées à la valeur 1 dans le système. L'ensemble de ces équations supplémentaires est noté $Fix_{\mathcal{G}_\kappa} = \bigcup_{c \in \mathcal{L}'_{\mathcal{G}_\kappa}} \{v_c = 1\}$ avec $\mathcal{L}'_{\mathcal{G}_\kappa}$ l'ensemble des connecteurs de $\mathcal{G}_\kappa$ ayant comme récepteurs un composant interactif. En outre, sur de grands systèmes, la résolution de ce problème donne souvent plusieurs solutions qui ne se valent pas du point de vue de la performance. C'est pour cela que nous privilégions celle qui maximise la performance de l'application et ce, en préservant le plus de jonctions du graphe initial. Cette contrainte supplémentaire est exprimée par la fonction objectif $Max_{\mathcal{G}_\kappa} = Maximize(\Sigma_{c \in \mathcal{J}_\mathcal{G}}(2^{g_c} \times (v_c))$ où $\mathcal{J}_{\mathcal{G}_\kappa}$ est l'ensemble des jonctions initialement présentes dans $\mathcal{G}_\kappa$ et g_c est un booléen prenant la valeur 1 si c est avec perte. Cette pondération des connecteurs avec perte a pour but, lorsque le choix se présente, de les préserver au détriment des nbBuffer afin de réduire les risques de saturation. Au final, le système linéaire à résoudre est $Eq_{\mathcal{G}_\kappa} \cup Fix_{\mathcal{G}_\kappa} \cup Max_{\mathcal{G}_\kappa}$.

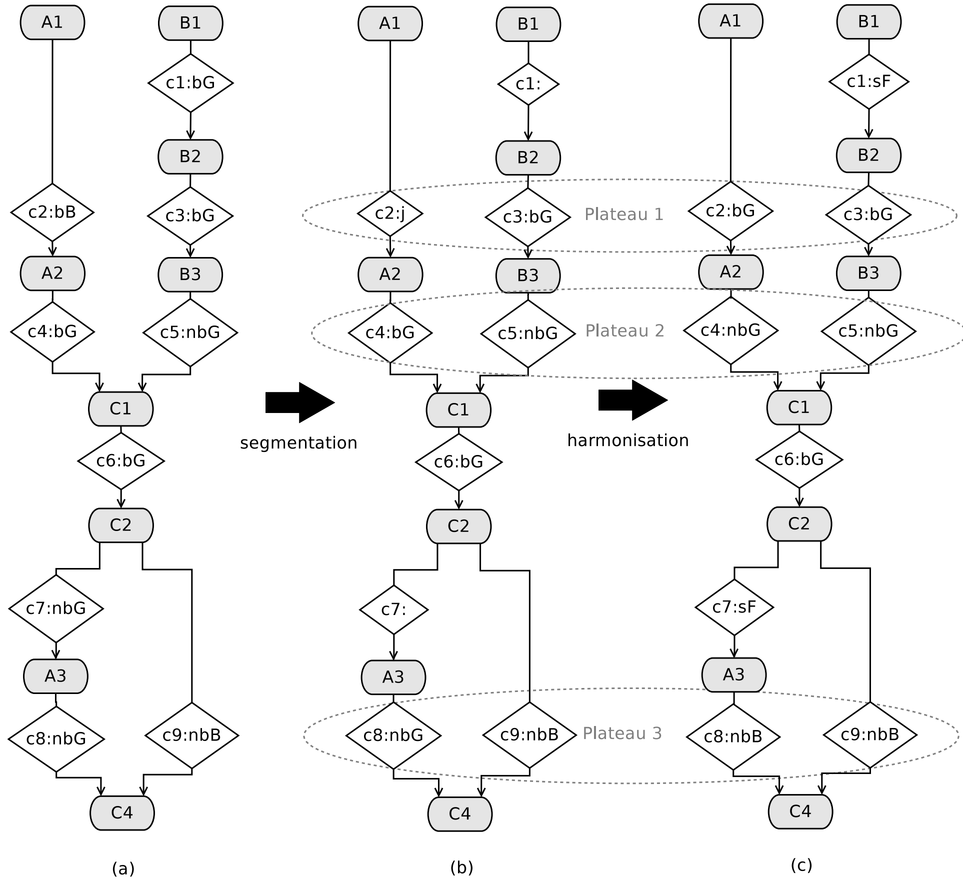

FIG. 7: Exemple de segmentation.

La figure 7b montre le résultat de la segmentation sur un exemple (figure 7a) où les chemins frères ont une portion commune. Dans cet exemple : $Eq_{\mathcal{G}_\kappa} = \{v_{c_2} + v_{c_4} + v_{c_6} + v_{c_7} + v_{c_8} = v_{c_1} + v_{c_3} + v_{c_5} + v_{c_6} + v_{c_9}\}$, $Fix_{\mathcal{G}_\kappa} = \{\}$ et $Max_{\mathcal{G}_\kappa} = Maximize(2 \times (v_{c_1} + v_{c_3} + v_{c_4} + v_{c_5} + v_{c_6} + v_{c_7} + v_{c_8}) + v_{c_9})$. Le résultat est : $v_{c_1} = 0, v_{c_2} = 1, v_{c_3} = 1, v_{c_4} = 1, v_{c_5} = 1, v_{c_6} = 1, v_{c_7} = 0, v_{c_8} = 1$ et $v_{c_9} = 1$. Une jonction a donc été créée (c_2) et deux autres supprimées (c_1 et c_7).

Des connecteurs synchrones remplacent les jonctions supprimées. Pour leur attribuer leur type, la règle est de privilégier le type sFIFO et sinon, lorsque la présence d'un composant émetteur interactif l'empêche, de choisir le type bBuffer.

3.2.2 Harmonisation des plateaux

La synchronisation de deux chemins repose sur la notion de *plateau*.

Définition 10 (Plateau) *Soient $\mathcal{G}$ un graphe d'application, $\kappa : A.o_{i \to C.i_1} = B.o_{j \to C.i_2}$ une contrainte de cohérence dans $\mathcal{K}_\mathcal{G}$ et P_1 et P_2 deux chemins frères dans ϕ_κ. P_1 et P_2 sont vus comme des successions de segments synchrones et de jonctions tels que $P_1 = (S_1^1, j_1^1, \ldots, j_1^{n-1}, S_1^n)$ et $P_2 = (S_2^1, j_2^1, \ldots, j_2^{m-1}, S_2^m)$, S_1^1 (respectivement S_2^1) débute par $A.o_i$ (respectivement $B.o_j$) et se termine par $C.i_1$ (respectivement $C.i_2$). Pour $k \leq n-1$ et $k \leq m-1$, on dit que les jonctions j_1^k et j_2^k sont de même niveau et on note $j_1^k \leftrightarrow j_2^k$. La clôture transitive réflexive de $\leftrightarrow$ est notée $\leftrightarrow^*$. Un* plateau *est l'ensemble des jonctions d'une classe d'équivalence de $\leftrightarrow^*$.*

Notons que d'après cette définition, si des plateaux de deux contraintes de cohérences ont au moins une jonction commune, ils sont fusionnés en un seul.

Sur la figure 6, les connecteurs c_1, c_3 et c_7 forment un plateau. Sur la figure 8, les couples de connecteurs $\{c_1, c_2\}$, $\{c_3, c_4\}$ et $\{c_w, c_x\}$ représentent, chacun, un plateau. Un plateau est le point d'entrée d'un ensemble de segments synchrones impliqués dans la même contrainte -ou dans des contraintes interdépendantes. C'est l'endroit où des messages circulant dans les différents chemins frères sont contrôlés les uns relativement aux autres. Donc, un plateau sera l'objet d'un motif de synchronisation destiné à établir ou à maintenir une cohérence.

Cette synchronisation des chemins par paliers induit une autre contrainte sur les chemins frères : les jonctions de même niveau doivent être du même type. Après que les nombres de jonctions des chemins de chaque cohérence aient été égalisés, il reste donc à fixer le type de chaque plateau. D'abord, les plateaux sont constitués d'après la définition 10. Les plateaux appartenant à différentes contraintes et ayant, au moins, une jonction commune sont groupés en un seul plateau tel que le précise la même définition. Si des jonctions appartenant à un même plateau sont de différents types, le système leur attribue un même type selon les règles suivantes :

1. nbBuffer s'il y a au moins un nbBuffer parmi elles ou s'il y a des jonctions nouvellement créées par la segmentation. Ce choix est dicté par la nécessité de respecter la contrainte de non perte sur les connexions. Notons ici que cette harmonisation ne sera pas possible si l'une des jonctions alimente un composant interactif. En effet, celle-ci doit nécessairement demeurer de type nbGreedy. L'établissement de la cohérence va donc échouer dans ce cas ;
2. nbGreedy sinon pour privilégier la performance.

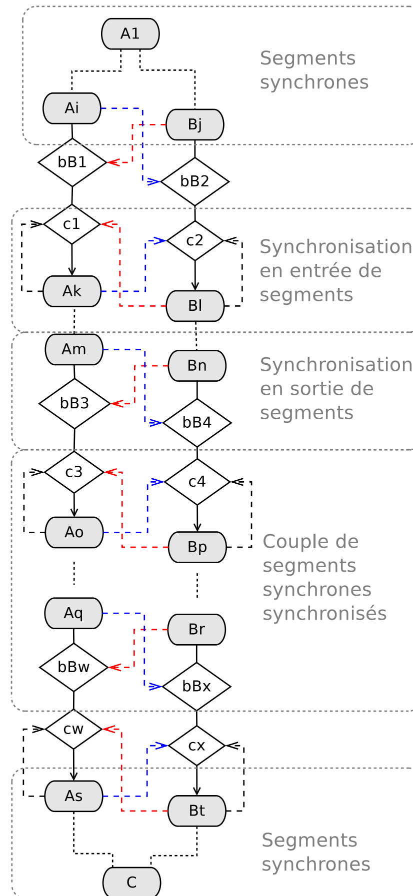

FIG. 8: Motif général pour la cohérence.

Notons que comme il n'y a que trois types de jonctions, si le plateau contient des jonctions différentes alors il y en a au moins une qui est nbBuffer ou nbGreedy.

La figure 7c montre le résultat de cette harmonisation sur les plateaux de la figure 7b.

3.2.3 Établissement de la cohérence

Une fois les plateaux constitués et harmonisés, les motifs de synchronisation visibles sur la figure 8 peuvent être automatiquement mis en place entre chaque couple de chemins frères dans $\mathcal{G}_\kappa$. Ces motifs sont de deux types :

- La synchronisation en entrée de segments ;
- La synchronisation en sortie de segments.

La synchronisation en entrée de segments (ou SES) est un motif de composition permettant d'assurer la cohérence en réception de données de deux segments synchrones.

Définition 11 (Synchronisation en entrée de segments) *Dans un graphe d'application $\mathcal{G}$, une synchronisation en entrée de segments est un motif de composition comprenant :*

- *deux segments synchrones S_1 et S_2 de respectivement k et l composants et se terminant par, respectivement, les composants C_1^k et C_2^l,*
- *un plateau constitué deux jonctions j_1 et j_2 de même type et précédant, respectivement, S_1 et S_2,*
- *un déclenchement croisé constitué des liens $(C_1^1.e, j_2.s)$ et $(C_2^1.e, j_1.s)$.*

Ce motif est noté $J * (S_1, S_2)$.

La SES assure que les jonctions j_1, j_2 sélectionnent leurs messages au même moment et qu'aucun nouveau message n'est accepté par les premiers composants des segments synchrones avant que tous deux ne soient prêts pour une nouvelle itération. Ce mécanisme fait l'hypothèse d'un temps de transfert nul d'un message ou d'un signal dans un lien. Si j_1 et j_2 sont non bloquants et ne contiennent pas de messages au moment où ils sont déclenchés, C_1 et C_2 reçoivent chacun un message vide. De manière générale, il est nécessaire que les deux jonctions aient le même comportement vis-à-vis de leur segment respectif et qu'elles soient donc de même type. C'est la raison pour laquelle nous avons effectué l'harmonisation des plateaux.

Observons que cette synchronisation entre les deux segments S_1 et S_2 reste valable si j_1 et j_2 sont, en plus, déclenchés par un même ensemble d'autres composants.

La synchronisation en sortie de segments (ou SSS) est un motif de composition permettant d'assurer la cohérence en émission de données de deux segments synchrones.

Définition 12 (Synchronisation en sortie de segments) *Dans un graphe d'application $\mathcal{G}$, une synchronisation en sortie de segments est un motif de composition impliquant :*

- *deux segments synchrones S_1 et S_2 de respectivement k et l composants et se terminant par, respectivement, les composants C_1^k et C_2^l,*
- *deux connecteurs bBuffer bB_1 et bB_2 succédant respectivement à S_1 et S_2 si $C_1^k \neq C_2^l$,*
- *un déclenchement croisé constitué des liens $(C_1^k.e, bB_2.s)$ et $(C_2^l.e, bB_1.s)$.*

Ce motif est noté $(S_1, S_2) * bB$.

Ce motif de composition permet que soit absorbé le décalage entre les messages que produisent les segments synchrones S_1 et S_2. Les bBuffer leur succédant sont automatiquement ajoutés. Comme ceux-ci sélectionnent au même moment leurs messages à émettre -lorsque les derniers composants C_1^k et C_2^l des segments synchrones ont tous deux terminé, ils les émettent simultanément. Comme la SES, ce mécanisme s'appuie sur le temps de transfert nul d'un message ou d'un signal à travers un lien.

Ici encore, cette synchronisation reste valable si bB_1 et bB_2 sont, en plus, déclenchés par un même ensemble d'autres composants.

Nous allons voir, à présent, qu'un motif de composition constitué d'une succession de SSS et de SES tel qu'illustré sur la figure 8 est à même de satisfaire une contrainte de cohérence $\kappa : A_1.o_{i \to C.i_1} = A_1.o_{j \to C.i_2}$ entre deux chemins $(P_1, P_2) = (S_1', S_2') * bB * [J * (S_1, S_2) * bB]^n * (S_1'', S_2'')$ tels que P_1 relie $A_1.o_i$ à $C.i_1$ et P_2 $A_1.o_j$ à $C.i_2$.

D'abord, les segments synchrones $S_1' = (A_1 \dots A_i)$ et $S_2' = (A_1 \dots B_j)$ garantissent, d'après la propriété 1, que pour tout message m_{A_i} émis par A_i, $it(m_{A_i}) = it(ori_{S_1'}(m_{A_i}))$ et que pour tout message m_{B_j} émis par B_j, $it(m_{B_j}) = it(ori_{S_2'}(m_{B_j}))$. Une SSS maintient la cohérence des couples de messages m_{A_i}, m_{B_j} sortants de ces deux segments. À ce stade,

$it(ori_{S'_1}(m_{A_i})) = it(ori_{S'_2}(m_{B_j}))$. Il s'agit ensuite de montrer que le reste du motif, composé de couples synchronisés de segments synchrones séparés par des plateaux, maintient cette cohérence jusqu'aux messages reçus par C.

Définition 13 (Segments synchrones synchronisés) *Dans un graphe d'application $\mathcal{G}$, le motif de composition $J * (S_1, S_2) * bB$ où S_1 et S_2 sont deux segments synchrones est appelé un couple de segments synchrones* sychronisés. *$[J * (S_1, S_2) * bB]^q$ exprime la composition de q segments synchrones synchronisés $J^1 * (S_1^1, S_2^1) * bB^1 * \cdots * J^q * (S_1^q, S_2^q) * bB^q$.*

Nous commençons par démontrer qu'un couple de segments synchrones synchronisés maintient la cohérence de deux flots de données qui le traversent. Cela revient à démontrer qu'à chaque itération, les deux segments acceptent deux messages ayant le même numéro d'itération et produisent, chacun et simultanément, un et un seul message. Formellement, soient M une série de messages acceptée ou émise par un segment, $|M|$ sa longueur et m^i son i^ème^ message. Un ensemble de séries de messages $\{M_1, \ldots, M_n\}$ est dit synchronisé si $|M_1| = \cdots = |M_n|$ et, $\forall i \in [1, |M_1|]$, $it(m_1^i) = \cdots = it(m_n^i)$.

Théorème 1 *Soit $\mathcal{G}$ un graphe d'application et $(S_1, S_2) = J * (s_1, s_2) * bB$ un couple de segments synchrones synchronisés dans $\mathcal{G}$. Si les séries de messages M_1 et M_2 stockés dans les jonctions j_1 et j_2 sont synchronisées, alors les couples de messages m_1 et m_2 stockés respectivement dans les connecteurs bB_1 et bB_2 sont tels que $it(m_1) = it(m_2)$ et $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2))$ lorsque les bBuffers sont déclenchés.*

Preuve 1 *Puisque les séries de messages stockés dans, respectivement, j_1 et j_2 sont synchronisées et que ces connecteurs sont déclenchés en même temps, les messages m_1^i et m_2^i qu'elles transmettent ont le même numéro d'itération k_1. Après ce déclenchement, les nouvelles séries de message contenues dans j_1 et j_2 sont encore synchronisées. Par construction, les premiers composants des deux segments commencent une nouvelle itération en même temps. Alors, leurs numéros d'itération sont toujours égaux et notés k_2. À cause de la propriété 1, nous savons que le numéro d'itération de chaque message émis par le dernier composant d'un segment synchrone est égal au numéro d'itération du message produit par son premier composant soit k_2. Puisque le message m_1 stocké dans bB_1 et le message m_2 stocké dans bB_2 ne sont rendus disponibles que quand les deux composants aux extrémités de S_1 et S_2 ont fini leurs itérations, nous obtenons $it(m_1) = it(m_2) = k_2$ et $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2)) = k_1$.*

Comme le laisse pressentir la figure 7, deux chemins frères peuvent s'intersecter avant leur destination commune, c'est à dire que des couples de segments synchrones peuvent avoir des parties communes. Dans la preuve du théorème 1 nous n'avons eu besoin d'aucune hypothèse de ce type. Nous pouvons cependant remarquer que ce cas peut permettre certaines simplifications quand la partie commune est en début ou en fin de la section synchronisée.

Dans le cas général, un couple de segments synchrones (S_1, S_2) est précédé d'un couple de jonctions J et suivi d'un couple de bBuffers bB. S'ils ont des portions communes entre leurs deux extrémités, ils restent synchrones. Dès lors, lorsqu'ils sont distincts à leurs deux extrémités, le motif de synchronisation $J * (S'_1, S'_2) * bB$ illustré par la figure 9a assure leur synchronicité d'après le théorème 1.

Si les sources de S_1 et de S_2 sont confondues, les deux segments sont précédés par une seule jonction j comme illustré dans la figure 9b. La spécification de la SES prend cette éventualité en compte. La SES est, ici, implicite car le même message alimente les deux segments,

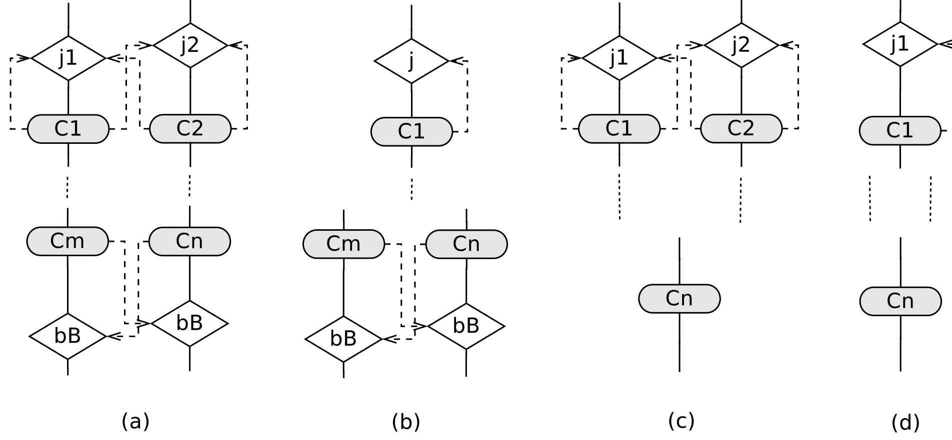

FIG. 9: Les quatre motifs de synchronisation entre deux segments sécants.

dans ce cas J est le couple (j, j). Le motif $J * (S'_1, S'_2)$ de la figure 9c illustre la situation inverse, lorsque les deux segments aboutissent au même composant. C'est la SSS qui est, alors, implicite puisque les deux segments émettent le même message résultat que bB soit présent ou non. Enfin, sur la figure 9d, les deux synchronisations sont implicites car les deux segments commencent par le même composant et se terminent par le même composant.

Le théorème suivant permet de montrer qu'une succession de segments synchrones permet de garantir la cohérence.

Théorème 2 *Soit $\mathcal{G}$ un graphe d'application et $(S_1, S_2) = [J * (S'_1, S'_2) * bB]^n$ deux chemins dans $\mathcal{G}$. Si les séries de messages M_1 et M_2 stockés dans les jonctions j_1^1 et j_2^1 du premier couple de segments sont synchronisées, alors tous les couples de messages m_1 et m_2 stockés, respectivement, dans les connecteurs bBuffer bB_1^n et bB_2^n des derniers segments synchrones synchronisés sont tels que $it(m_1) = it(m_2)$ et $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2))$ quand ces bBuffers sont déclenchés.*

Preuve 2 *D'après le théorème 1, si les séries de messages stockés dans les jonctions sont synchronisées, alors les messages transmis par les bBuffer ont le même numéro d'itération. Par conséquent, les séries de messages stockés dans les jonctions suivantes sont synchronisées. Une simple induction sur n prouve ce théorème.*

Nous pouvons alors affirmer que la construction $(P_1, P_2) = (S'_1, S'_2) * bB * [J * (S_1, S_2) * bB]^n * (S''_1, S2''_2)$ réalise bien une cohérence entre les deux chemins P_1 et P_2.

Théorème 3 *Soient $(P_1, P_2) = (S'_1, S'_2) * bB * [J * (S_1, S_2) * bB]^n * (S''_1, S2''_2)$ deux chemins frères relativement à la contrainte de cohérence $\kappa : A.o_{1 \to C.i_1} = B.o_{2 \to C.i_2}$). Soient m_1 et m_2 deux messages lus par C à la même itération sur, respectivement, les ports $C.i_1$ et $C.i_2$. Alors m_1 et m_2 vérifient que $it(ori_{P_1}(m_1)) = it(ori_{P_2}(m_2))$.*

Preuve 3 *Les chemins P_1 et P_2 se terminent par un segment synchrone (S''_1 et S''_2) par conséquent d'après la propriété 1, la cohérence est déjà établie entre eux.*

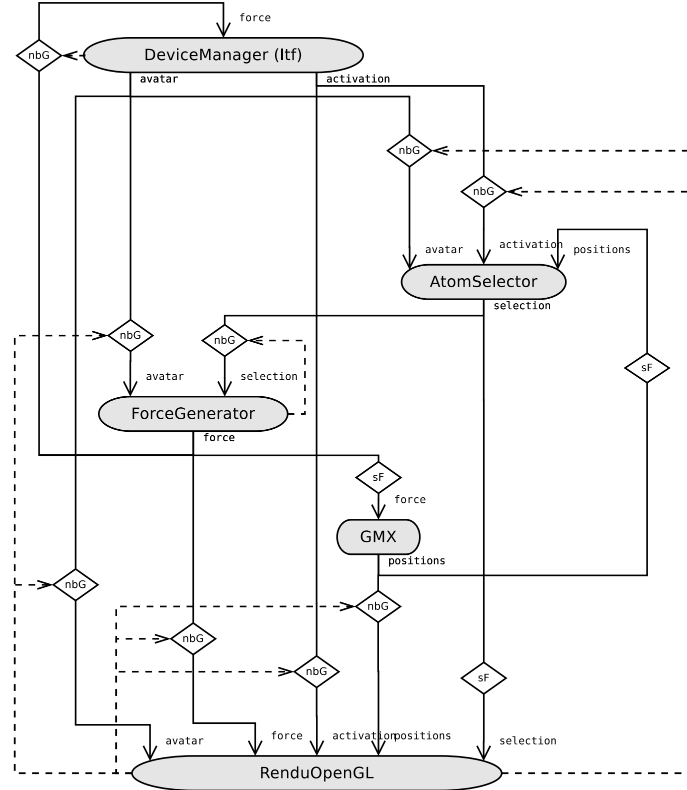

FIG. 10: Graphe d'application généré pour FVnano.

L'application de ce motif à tous les couples de chemins frères dans l'ensemble ϕ_κ d'une contrainte κ rend cette cohérence conforme à la définition 4.

La figure 10 montre le graphe d'application final spécifié dans la figure 4 après mise en place du motif pour réaliser la cohérence

$$\kappa : DM.avatar_{\rightarrow RO.avatar} = DM.avatar_{\rightarrow RO.selection}$$

On voit qu'une factorisation des liens de déclenchement peut également être opérée en fin de traitement. Par exemple, initialement, RenduOpenGL déclenche AtomSelector (connexion sFIFO) qui déclenche le nbGreedy qui l'alimente. Cela revient à ce que RenduOpenGL déclenche directement ce nbGreedy. Il en va de même pour les connecteurs nbGreedy qui alimentent ForceGenerator.

On peut noter que notre construction fournit un graphe final d'application qui contient les mêmes composants que le graphe de spécification. De plus, le nombre de connecteurs est strictement inférieur à deux fois le nombre de liens du graphe de spécification (un connecteur par lien, plus les connecteurs de type bBuffer du motif SSS).

4 Application à une simulation de dynamique moléculaire

Dans cette section, nous testons notre méthode de construction par l'implémentation de l'application de la figure 10. Il s'agit d'une version simplifiée d'une application de dynamique moléculaire interactive développée dans le cadre du projet ANR FVnano. La figure 11 montre une capture d'écran du rendu produit par le composant d'affichage.

Nous utilisons l'intergiciel FlowVR Allard et al. (2004) comme modèle d'exécution. Il permet de créer et d'exécuter des applications C++ haute-performance distribuées. Le graphe d'application obtenu a été implémenté et exécuté sur une machine équipée d'un processeur Intel Core2 Duo 2x2.5 Ghz. La molécule simulée est celle visible sur la figure 11. Elle contient près de 3000 atomes. Nous avons comparé les performances de cette application à celles d'une version antérieure non basée sur notre modèle et construite manuellement dans le but de réaliser la même cohérence. Le sous-graphe de cohérence dans cette version est représenté par la figure 12. Cette version n'étant pas basée sur notre modèle de composants, ses connecteurs n'ont pas d'équivalent exact dans notre modèle. Par exemple, le filtre Greedy représenté renvoie, en l'absence de nouveaux messages, le dernier message envoyé plutôt qu'un message vide. Le connecteur "F" est, lui, une simple FIFO. La représentation de la coordination dans ce sous-graphe est cependant fidèle à la réalité.

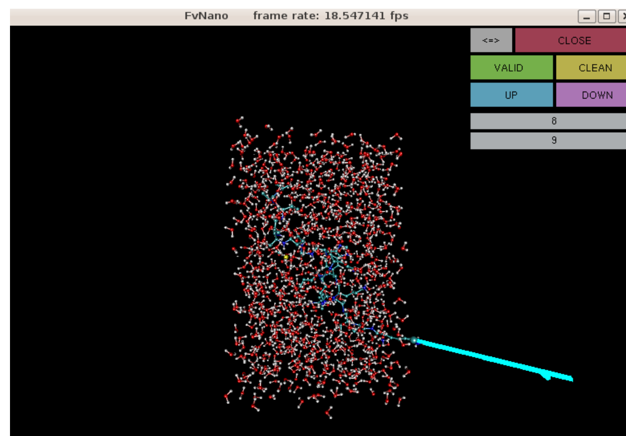

FIG. 11: Capture d'écran de l'application FVnano.

On observe sur cette construction manuelle que la cohérence, d'après notre définition 4, n'est pas satisfaite. La raison principale est le nombre différent de connecteurs Greedy présents sur les trois chemins. Toutefois, cette construction a jusqu'à maintenant été utilisée car elle fournit un résultat visuel satisfaisant.

Le fait que chaque module ne contrôle pas systématiquement le connecteur qui l'alimente introduit également des risques de saturation dans la construction manuelle. Ainsi, le composant RenduOpenGL peut provoquer la saturation des composants ForceGenerator et AtomSelector s'il est plus rapide qu'eux. De la même manière, ForceGenerator peut saturer GMX. Ce dernier cas s'est d'ailleurs vérifié lors de nos tests et nous avons observé une chute progressive

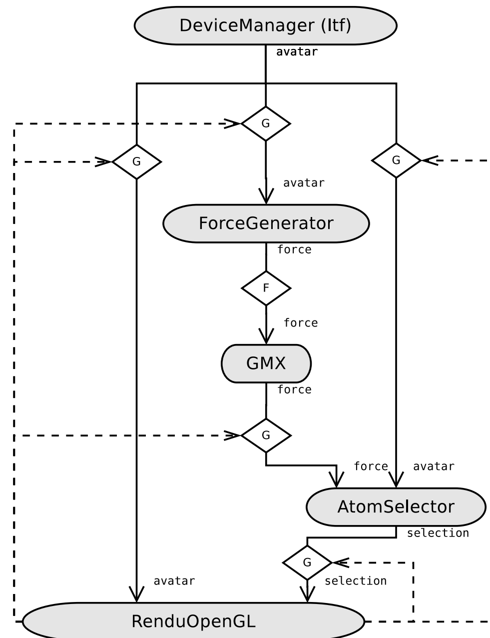

FIG. 12: Sous-graphe à partir de l'application manuellement construite.

de la performance de GMX à cause de sa saturation. Un module FlowVR saturé finit ensuite par se bloquer au bout d'une certaine quantité de messages accumulés en entrée.

Pour estimer la qualité de notre construction, nous nous intéressons au respect effectif de la cohérence et à la performance de certains composants comme la simulation ou la visualisation. On peut remarquer que le taux de perte des messages n'est pas facile à interpréter de manière absolue. Il peut être très différent d'une jonction à l'autre sans nuire pour autant au comportement global de l'application. Il est clair que cette perte est fortement liée à la vitesse relative du composant émetteur par rapport au composant récepteur. Par contre notre construction garantit que un maximum de messages est traité en évitant toute saturation. En effet, dès que les composants récepteurs sont disponibles ils acceptent un nouveau message.

Dans l'application basée sur notre modèle, la cohérence est obtenue grâce à une SES.

Au niveau de la performance, comme le montre la figure 10, les modules AtomSelector, ForceGenerator, GMX et RenduOpenGL sont synchronisés. Leurs fréquences s'alignent donc sur celle du plus lent parmi eux, GMX en l'occurrence. Ce module n'étant plus saturé, sa fréquence maximale remonte à 91 it/s soit une augmentation de 10.3% par rapport à celle la version construite manuellement.

5 Conclusion

La cohérence -en plus de la performance et de la simplicité- est un critère majeur pour les scientifiques au moment de construire leurs propres logiciels. A notre connaissance, la co-

hérence telle qu'entendue dans ce papier et sa réalisation par la modification automatique de graphes n'a pas encore été traitée. Nous avons démontré qu'une application relativement complexe pouvait être automatiquement construite à partir d'une spécification utilisateur exprimée en termes de contraintes de communication et de cohérence. Non seulement les applications générées assurent-elles la cohérence des données entrantes où cela est requis mais elles garantissent également l'exécution la plus sûre en termes de saturation ou d'écrasement non souhaité de données.

Hormis l'application de dynamique moléculaire présentée dans cet article, notre méthode a également été éprouvée sur une application de rendu de terrain comportant deux contraintes de cohérence combinées. Ainsi, notre approche s'accommode tout à fait d'applications plus complexes nécessitant plusieurs cohérences simultanément. La taille des applications que nous avons rencontrées dans le domaine de la visualisation scientifique interactive ne dépasse pas une vingtaine de modules et une centaine de liens ce qui rend la question de la complexité de la construction secondaire. On peut cependant noter que les deux phases les plus coûteuses sont la recherche des graphes de cohérence (qui revient à rechercher tous les chemins sans cycle entre deux points d'un graphe) et la résolution du système linéaire dont le nombre de variables est inférieur au nombre de liens dans le graphe de spécification.

Il serait intéressant de compléter nos expérimentations en testant un plus grand nombre d'applications. Afin de faciliter et de systématiser ces expérimentations, nous sommes en train de développer un simulateur à base de composants factices dont on peut contrôler la fréquence. Ce simulateur permettra de tester de nombreuses configurations avec des contraintes de cohérence pouvant être complexes.

Dans l'exemple de la simulation moléculaire construite manuellement, nous avons remarqué que la cohérence implémentée était plus souple que celle que nous avons définie mais était suffisante pour les utilisateurs. Nous travaillons donc à la définition de cohérences moins strictes qui pourraient garantir un meilleur compromis entre performance et cohérence. Par exemple, nous souhaiterions relâcher l'égalité stricte des numéros d'itérations ou travailler sur des informations autres que ce numéro d'itération. En effet, la plupart des simulations scientifiques sont basées sur un temps de simulation qui pourrait être exploité pour de nouveaux types de cohérence. Ces informations devront cependant respecter certaines propriétés garanties par les numéros d'itération (croissance au cours du temps, une certaine continuité).

Références

Allard, J., V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, et S. Robert (2004). FlowVR : a middleware for large scale virtual reality applications. In *Euro-par 2004 Parallel Processing*, pp. 497–505. Springer.

Allard, J., J.-D. Lesage, et B. Raffin (2010). Modularity for Large Virtual Reality Applications. *Presence : Teleoperators and Virtual Environments 19*(2), 142–161.

Arbab, F., T. Chothia, S. Meng, et Y. Moon (2007). Component connectors with QoS guarantees. In *Coordination Models and Languages*, pp. 286–304. Springer.

Barseghian, D., I. Altintas, M. B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. T. Borer, et E. W. Seabloom (2010). Workflows and extensions to the

Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics 5*(1), 42–50.

Bouziane, H., C. Pérez, et T. Priol (2008). A software component model with spatial and temporal compositions for grid infrastructures. *Euro-Par 2008–Parallel Processing*, 698–708.

Churches, D., G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, et I. Wang (2006). Programming scientific and distributed workflow with Triana services. *Concurrency and Computation : Practice and Experience 18*(10), 1021–1037.

Deelman, E., D. Gannon, M. Shields, et I. Taylor (2009). Workflows and e-Science : An overview of workflow system features and capabilities. *Future Generation Computer Systems 25*(5), 528–540.

Delalande, O., N. Ferey, B. Laurent, M. Gueroult, B. Hartmann, et M. Baaden (2010). Multiresolution approach for interactively locating functionally linked ion binding sites by steering small molecules into electrostatic potential maps using a haptic device. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing 215*, 205–15.

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.

McAdam, R. J. (2010). Continuous interactive simulation : Engaging the human sensory-motor system in understanding dynamical systems. *Procedia Computer Science 1*(1), 1691–1698.

Mulder, J. D., J. J. van Wijk, et R. van Liere (1999). A survey of computational steering environments. *Future Generation Computer Systems 15*(1), 119–129.

Pautasso, C. et G. Alonso (2006). Parallel computing patterns for Grid workflows. In *2006 Workshop on Workflows in Support of Large-Scale Science*, pp. 1–10. IEEE.

Weinstein, D., S. Parker, J. Simpson, K. Zimmerman, et G. Jones (2005). Visualization in the scirun problem-solving environment. *The Visualization Handbook*, 615–632.

Wierse, A. (1996). Collaborative visualization based on distributed data objects. *Database Issues for Data Visualization*, 208–219.

Yildiz, U., A. Guabtni, et A. Ngu (2009). Towards scientific workflow patterns. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, New York, New York, USA, pp. 1–10. ACM.

Summary

This paper addresses the use of component-based development to build interactive scientific visualization applications. Our overall approach is to make this programming technique more accessible to non-computer-scientists. Therefore, we present a method to, out of constraints given by the user, automatically build and coordinate the dataflow of a real-time interactive scientific visualization application. This type of applications must run as fast as possible while preserving the accuracy of their results. These two aspects are often conflicting, for example when it comes to allowing message dropping or not. Our method aims at automatically finding the best balance between these two requirements when building the application up.