Un modèle pour la composition d'applications de visualisation et d'interaction continue avec des simulations scientifiques

Ahmed Turki

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

THÈSE présentée par :

Ahmed TURKI

soutenue le : 8 mars 2012

pour obtenir le grade de : Docteur de l'Université d'Orléans

Discipline/ Spécialité : Informatique

Un modèle pour la composition d'applications de visualisation et d'interaction continue avec des simulations scientifiques

THÈSE dirigée par :

Sébastien LIMET — Professeur, Université d'Orléans

Co-encadrée par :

Sophie ROBERT — Maître de conférences, Université d'Orléans

RAPPORTEURS :

Benoît BAUDRY — Chargé de recherche, INRIA Rennes

Christian PEREZ — Directeur de recherche, INRIA Rhône-Alpes

JURY :

Benoît BAUDRY	Chargé de recherche, INRIA Rennes
Christian PEREZ	Directeur de recherche, INRIA Rhône-Alpes
Bruno RAFFIN	Chargé de recherche, INRIA Rhône-Alpes
Mirian HALFELD FERRARI ALVES	Professeur, Université d'Orléans
Sébastien LIMET	Professeur, Université d'Orléans
Sophie ROBERT	Maître de conférences, Université d'Orléans

Remerciements

Ayant entamé cette thèse sans être familier de la programmation par composants, je suis fier que Christian Perez et Benoît Baudry aient, au final, apporté leur caution de spécialistes à ses résultats en acceptant de la rapporter.

Je remercie aussi et surtout mes encadrants Sébastien Limet et Sophie Robert pour m'avoir appris toutes les étapes du travail de recherche dont je ne savais rien. Je leur suis également reconnaissant de m'avoir offert un sujet aussi original ainsi que pour leur constante bienveillance à mon égard.

Je tiens ensuite à remercier Abdelali Ed-Dbali, responsable du département d'informatique de l'UFR Sciences, pour m'avoir permis de mener à bien ma mission d'enseignement au sein de son département. Je suis, par ailleurs, très heureux d'avoir exercé dans un département et un laboratoire aussi accueillants.

De la même manière, j'adresse un amical salut à mes camarades du FSK Orléans dont j'ai apprécié la compagnie pendant mes heures de loisir. Merci particulièrement à ceux qui se sont déplacés pour assister à ma soutenance.

Je souhaite également saluer tous les collègues du projet FVnano avec lesquels j'ai eu plaisir à collaborer : Bruno Raffin, Marc Baaden, Jean-Philippe Nominé, Jean-Denis Lesage, Nicolas Férey, Olivier Delalande, Antoine Vanel, Matthieu Chavent, Alex Tek et Marc Piuzzi. J'espère vivement retravailler avec vous un jour.

Enfin, les derniers remerciements vont à mes parents pour leur continuel soutien et à mes soeurs pour leur gentillesse.

Cette thèse a été financée par l'Agence Nationale de la Recherche (ANR) à travers le projet FVnano.

Table des matières

Liste des définitions

Introduction

Contexte

La simulation informatique fait aujourd'hui partie intégrante de tout projet scientifique académique ou industriel. Les avancées dans la modélisation mathématique d'objets techniques et de phénomènes physiques donnent aux scientifiques la possibilité de (re)produire, sur ordinateur, des expériences de plus en plus complexes. Par rapport à l'expérience en conditions réelles, la simulation a permis d'augmenter le nombre d'essais en économisant les matières premières et en annulant les risques et les temps de mise en place. Le temps de simulation, lui, est inversement proportionnel à la puissance des machines sur lesquelles la simulation est exécutée. Il est réduit par chaque nouvelle génération de processeurs ce qui permet aux scientifiques de répéter leurs expériences un plus grand nombre de fois.

La parution d'une nouvelle génération de processeurs rend également les précédentes financièrement plus abordables. Les laboratoires de recherche, établissements très demandeurs en puissance calcul, sont alors de plus en plus nombreux à s'équiper de grappes. Une grappe (ou *ferme de calcul* ou *cluster*, en anglais) est un regroupement, en réseau, d'ordinateurs appelés *nœuds* dans ce contexte et situés dans un même local. Une grappe est une ressource matérielle idéale pour les applications parallèles ou modulaires. Les premières sont des programmes conçus pour être exécutés en plusieurs "copies" (on parle d'*instances* ou de *threads*, en anglais) et simultanément sur différents jeux de données. Les secondes sont des applications formées de sous-programmes, les *modules* ou *composants*, exécutant chacun une tâche élémentaire de l'ensemble. L'utilisateur disposant d'une grappe peut ainsi répartir sur ses nœuds les instances d'une application parallèle ou les composants d'une application modulaire. Le but de cet exercice, appelé *déploiement*, est de fournir à chaque programme ou sous-programme la puissance de calcul suffisante pour qu'il s'exécute en un temps raisonnable. Par ailleurs, une application peut tout à fait être à la fois modulaire et parallèle lorsque certains de ses composants sont parallélisables.

Cette disponibilité en ressources de calcul et surtout l'émergence de l'informatique graphique a également eu pour effet de démocratiser la visualisation scientifique. Jusqu'à la fin du XX^{e} siècle, les résultats d'une simulation informatique étaient produits sous forme de tableaux de valeurs numériques ou, au mieux, de courbes graphiques. Aujourd'hui, il est courant que ces résultats soient représentés par une image en deux ou trois dimensions des modèles étudiés. L'expérience virtuelle se voit ainsi complétée par sa représentation physique -bien que virtuelle elle aussi- se prêtant ainsi au mode d'interprétation qu'aurait le scientifique devant une expérience *in situ*. La visualisation scientifique est particulièrement prisée dans l'étude de systèmes dynamiques tels que la mécanique des fluides [51,89] ou la dynamique moléculaire [12,20]. En outre, l'usage de la visualisation se limite rarement à une reproduction réaliste de l'objet étudié. Le contrôle qu'elle permet sur le rendu graphique est, en effet, bien souvent l'occasion d'expliciter des grandeurs imperceptibles à l'œil nu : nanoparticules, chaleur, etc. Enfin, la visualisation scientifique peut être *interactive*. Elle

l'est lorsque l'homme est capable d'agir sur la simulation pendant qu'elle se déroule. La visualisation constitue alors le premier vecteur de perception des effets de ses actions. D'autres interfaces (sonores, haptiques, etc.) peuvent compléter le dispositif. Ainsi, la force et le retour de force peuvent parfois être utiles pour comprendre les lois qui régissent un système dynamique [66].

Comme toute application de calcul intensif, beaucoup de simulateurs scientifiques sont aujourd'hui parallèles. Il existe également des logiciels, appelés *environnements de composition* ou *frameworks* en anglais, pour construire de complexes applications scientifiques modulaires en couplant plusieurs programmes de calcul. Cette manière de développer des applications par composition promeut la collaboration entre chercheurs et la réutilisation de codes existants et éprouvés. L'architecture modulaire de l'application laisse aussi la possibilité de l'utiliser sur une grappe pour en accroître les performances.

Chaque environnement de composition peut avoir sa propre définition du composant, de son fonctionnement et de sa manière de communiquer avec les autres composants. Cette spécification constitue le *modèle* qui lui est sous-jacent. Entre deux modèles de composants, on observe souvent une base commune et, ensuite, certaines spécificités dictées par le(s) domaine(s) d'application visé(s). Cependant, aujourd'hui encore, une majorité de scientifiques investissent un temps conséquent dans le développement de solutions de couplage ad-hoc et peu pérennes. Cela est principalement dû à une possible difficulté dans la prise en main des outils de composition. Pour satisfaire leurs besoins, les concepteurs d'environnements de composition génériques doivent faire face à plusieurs problématiques métier : faire en sorte que les codes très hétérogènes qui forment les composants s'échangent des données compatibles, s'assurer que l'utilisateur puisse organiser l'exécution des tâches comme il le souhaite, veiller à ce que l'interface de l'outil soit compréhensible par les scientifiques auxquels elle se destine, etc. Quelques environnements pour la composition d'applications de calcul scientifique répondent à ces exigences.

L'intérêt croissant pour la visualisation scientifique interactive dans les sciences expérimentales laisse présager l'apparition de besoins en composition pour ce type d'applications également. Cette composition doit prendre en compte des contraintes de performance et de cohérence plus fortes qu'elles ne sont dans la composition de codes de calcul uniquement. À ces contraintes, les outils actuels pour la composition d'applications scientifiques ne répondent, à notre connaissance, pas encore.

Objet et organisation du mémoire

Dans ce mémoire, nous définissons un modèle de composants destiné aux applications de visualisation scientifique interactive. Dans ce modèle, la coordination entre les composants est au centre des préoccupations. Celle-ci pouvant devenir complexe dans de grandes applications, nous proposons également une méthode destinée à automatiquement établir cette coordination à partir de contraintes formulées par l'utilisateur.

Dans la première partie du mémoire, nous faisons l'état de l'art des deux domaines à la croisée desquels se trouve cette thèse : la visualisation scientifique (chapitre 1) et la programmation par composants (chapitre 2). Au chapitre 3, nous comparons quelques exemples représentatifs d'environnements pour la composition d'applications scientifiques. Le but de cette étude est d'évaluer la convenance des modèles actuels à la composition d'applications de visualisation interactive et, éventuellement, d'en déduire les propriétés adéquates pour un nouveau modèle. La partie II contient notre contribution à la composition d'applications de visualisation interactive. Le chapitre 4 décrit notre modèle de composants pour ce type d'applications. Le chapitre 5, lui, détaille les étapes pour auto-

matiquement construire une application répondant à ce modèle à partir d'une spécification fournie par l'utilisateur. Nous y proposons, en particulier, une méthode pour automatiquement adapter les connexions inter-composants dans le but de concilier performance et cohérence dans l'application. Ensuite, la partie III est dédiée à l'évaluation de notre modèle et schémas de coordination sur des applications C++ dont certaines sont distribuées. Enfin, dans la dernière partie, nous évaluons notre approche et déterminons différentes perspectives qu'elle permet d'entrevoir.

Première partie

État de l'art

Chapitre 1

Visualisation scientifique interactive

Sommaire

1.1 Origines

FIGURE 1.1 – Vues de côté et de dessus de l'écoulement de l'eau dans un bac par Léonard De Vinci.

La figure 1.1, extraite du livre *Frontiers of Scientific Visualization* de Pickover et Teksbury [76], est l'une des plus anciennes preuves d'utilisation de la visualisation, ici dans l'étude de l'hydrodynamique. L'auteur du dessin, l'inventeur et artiste Léonard De Vinci (1452-1519), commente ses observations par ces mots :

> *Observez le mouvement de la surface de l'eau qui ressemble à celui du cheveu lequel a deux mouvements. L'un est causé par le poids du cheveu, l'autre par la direction des boucles. L'eau a donc un mouvement tourbillonnant dont une partie est due au courant principal et l'autre au mouvement aléatoire inverse.*

D'après Lumley [61], De Vinci aurait ainsi mis en évidence le principe de décomposition de la turbulence, formalisé par Reynolds [79] quatre siècles plus tard. Ce n'est qu'un exemple parmi d'autres des phénomènes physiques établis aux XIX^{e} et XX^{e} siècles et sur lesquels De Vinci aurait communiqué des théories au moyen de la visualisation [34].

Historiquement, la visualisation dans la science revêt donc un sens très large qui est l'illustration de l'information scientifique dans un but premier de communication. Les cartes géographiques relèvent de la visualisation. Un autre exemple, plus original, est la surface thermodynamique en argile que sculpta James Clerk Maxwell (figure 1.2) en 1874. L'année précédente, Maxwell avait pris connaissance des travaux de Josiah Willard Gibbs sur la relation entre le volume, l'entropie et l'énergie d'une substance [36]. Enthousiasmé par ce résultat, il voulut pallier au manque de schémas dans la publication de son confrère en en réalisant son propre diagramme en trois dimensions [64]. Maxwell préfigura alors la visualisation scientifique telle qu'on l'entend de nos jours.

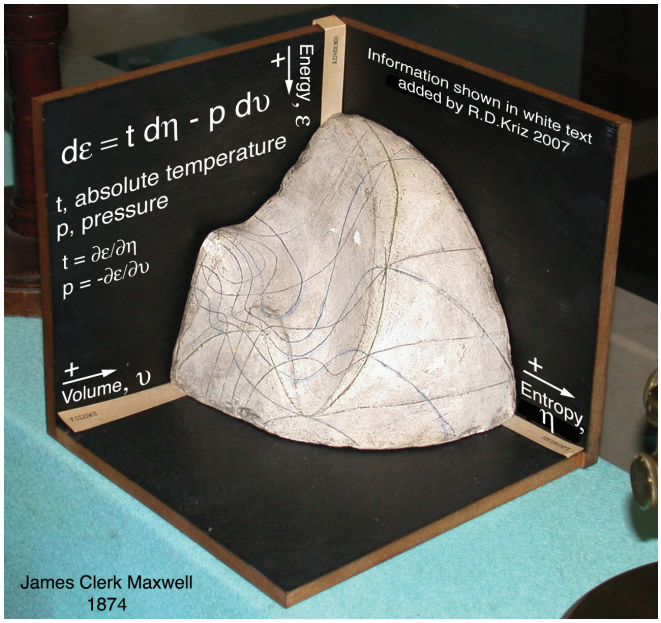


FIGURE 1.2 – Exposition d'un moulage en plâtre de la sculpture de Maxwell, Cambridge.

La popularisation de l'informatique graphique en deux (2D) et trois (3D) dimensions et son association à la simulation scientifique donna, à la fin des années 1980, son sens informatique actuel à la visualisation scientifique [67]. Adjointe à la simulation sur ordinateur, la visualisation joue un rôle bien avant la phase de publication, au moment de l'interprétation des résultats. En effet, contrairement aux expériences réelles, les simulations ne produisent que des résultats sous forme numérique. Disposer d'une représentation picturale de ces résultats accélère leur compréhension par le chercheur, en particulier lorsqu'ils sont fonction du temps.

1.2 Différentes formes d'interactivité

Aujourd'hui encore, une confusion règne quant à la définition de la visualisation scientifique interactive. Elle est due au fait que cette appellation ne permet pas de connaître le champ d'application exact de l'interactivité au sein de l'application. S'agit-il uniquement de personnaliser le mode d'affichage des images à l'écran, de naviguer dans la scène 3D pendant ou après le calcul ou bien est-ce une interactivité plus intrusive, impliquant une interaction avec la simulation alors qu'elle se déroule ? Pour correctement distinguer devant quel type d'application on se trouve, il faudrait remonter à trois disciplines de l'informatique.

1.2.1 Le pilotage de simulations

Le pilotage (*steering*, en anglais) se définit par le contrôle interactif d'un calcul durant son exécution [69]. On parle de pilotage de simulation lorsque ce calcul vise à modéliser un système naturel, technique voire social.

La visualisation peut faire partie du dispositif de pilotage sans nécessairement y être un vecteur d'interaction avec la simulation. Il est, en effet, plus commun que l'utilisateur pilote la simulation par commande textuelle [46,86] ou via une boîte de dialogue [50,62,78,88]. Par définition, l'interaction induite par ces deux interfaces "discrètes" est ponctuelle dans le temps. Son but est d'ajuster, à la volée, les paramètres de la simulation afin de la diriger vers une solution ou de l'en éloigner. Elle favorise ainsi les méthodes de recherche dites par affinage de la zone d'intérêt [62]. La finalité du pilotage est donc, de manière générale, d'élucider plus vite la relation paramètres-résultat, sans avoir à systématiquement redémarrer l'application.

Le pilotage comprend également le contrôle de l'exécution à savoir sa vitesse, la capacité de mettre en pause l'application, de la redémarrer ou de l'arrêter complètement [46,80].

1.2.2 La visualisation

La visualisation scientifique est définie, techniquement, par le procédé qui transforme des données scientifiques en éléments picturaux destinés à être stockés ou immédiatement affichés. Est aussi inclus dans la définition initiale [67] le procédé inverse transformant des images, grâce à la vision par ordinateur, en données dans le but d'être soumises à un calcul scientifique. La visualisation scientifique est également définie par ses objectifs qui sont la compréhension, la communication, l'enseignement et la vulgarisation [23].

Aucune synchronicité avec la simulation n'est imposée par cette définition. Toutefois, l'avantage de la visualisation pour le pilotage est mis en avant dès le départ. Une définition de "calcul visuel interactif" est même avancée comme étant le fait de pouvoir manipuler la représentation visuelle des données pendant leur traitement [23]. Le pilotage n'est alors présenté que comme sa forme la plus sophistiquée, les autres étant :

- Le monitoring : suivi en direct de la simulation sans interaction possible avec elle ;
- Le post-processing : visualisation après calcul, éventuellement avec une interaction de type navigation.

1.2.3 La réalité virtuelle

La définition technique de la réalité virtuelle est [33] :

Définition 1 (Réalité virtuelle) *La réalité virtuelle est un domaine scientifique et technique exploitant l'informatique et des interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudo-naturelle par l'intermédiaire de canaux sensori-moteurs.*

Afin de favoriser l'immersion de l'utilisateur dans l'espace virtuel, ce dernier est obligatoirement représenté en trois dimensions. Aucun niveau de réalisme ne lui est, en revanche, imposé. Cependant, dans la majorité des cas, il consiste en une représentation "suffisamment fidèle" d'un environnement réel enrichi de symboles visuels.

La différence majeure entre visualisation et réalité virtuelle est que cette dernière implique nécessairement une interaction temps réel avec la simulation. Bien que ce soit une

caractéristique qu'elle partage avec le pilotage, cette interaction est également continue dans le temps et sa finalité [33] en sciences diffère de celle du pilotage :

Définition 2 (Finalité de la réalité virtuelle) *La finalité de la réalité virtuelle est de permettre à une personne (ou à plusieurs) une activité sensori-motrice et cognitive dans un monde artificiel, créé numériquement, qui peut être imaginaire, symbolique ou une simulation de certains aspects du monde réel.*

La réalité virtuelle scientifique ne s'applique qu'à des systèmes ayant des propriétés physiques et soumis à des forces internes et externes. Elle consiste à créer une boucle sensorimotrice dans laquelle les mouvements de l'utilisateur sont traduits en paramètres du système et où l'état du système est, en retour, traduit par un signal sensoriel visuel, auditif ou haptique. Le but d'un tel dispositif, déjà évoqué sous le nom de "simulation interactive continue" (*continuous interactive simulation*, en anglais) [66], est de permettre à l'utilisateur de découvrir le comportement d'un système dynamique de manière plus intuitive, grâce à son système sensitif. La relation paramètres-résultat est, ici, plaquée sur le rapport action-réaction.

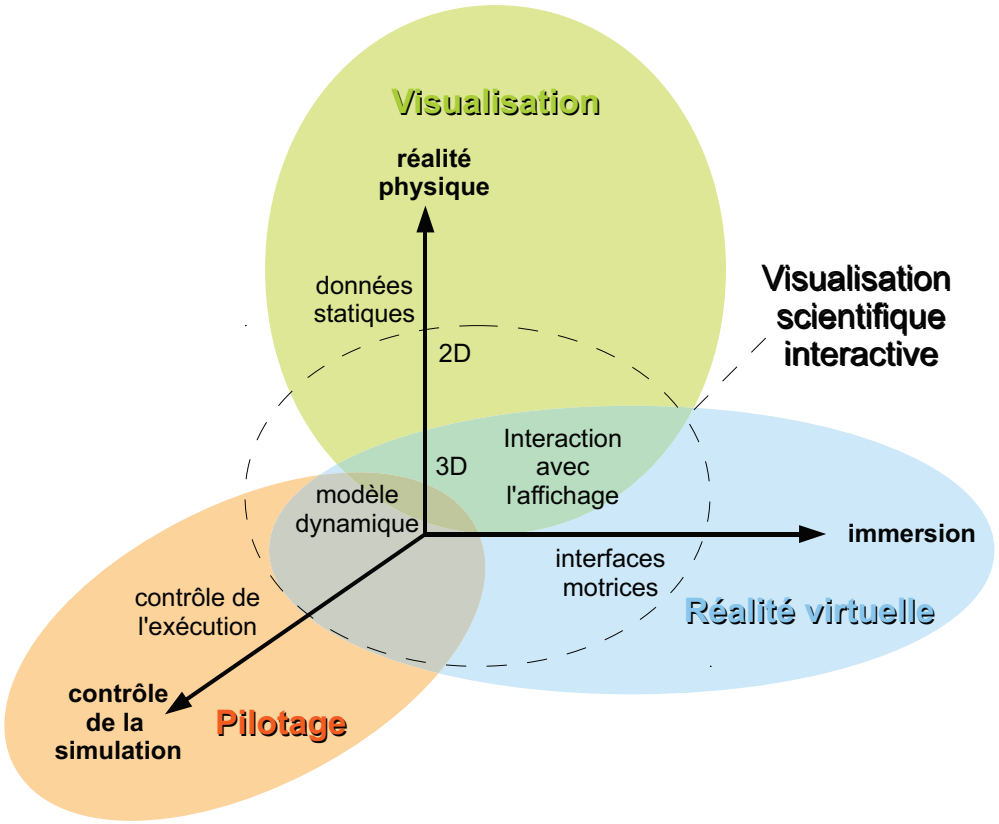


FIGURE 1.3 – Caractéristiques du pilotage, de la visualisation et de la réalité virtuelle.

1.3 Conclusion

Dans ce chapitre, nous avons souhaité souligner la grande hétérogénéité de la visualisation scientifique interactive en tant que discipline scientifique. La figure 1.3 tente de situer

la visualisation, le pilotage et la réalité virtuelle les uns par rapport aux autres. Les trois disciplines ont à la fois des points communs et se caractérisent, chacune, par une fonction essentielle qui se trouve être mineure ou totalement absente dans les autres. Il s'agit du contrôle de la simulation pour le pilotage, de la représentation d'un modèle physique réel pour la visualisation et de l'immersion de l'utilisateur dans la réalité virtuelle.

Le cercle en pointillés situe notre acception de la visualisation scientifique interactive qui emprunte aux trois disciplines. Les applications scientifiques auxquelles nous nous intéressons s'articulent autour de simulations physiques continues sur lesquelles l'utilisateur peut agir par le biais d'un périphérique moteur. Concrètement, il pourrait s'agir d'une simulation de dynamique moléculaire dans laquelle l'utilisateur, à l'aide d'un bras haptique, tirerait ou pousserait des atomes et percevrait, visuellement et/ou par retour de force, la réaction de la molécule.

La suite de cette partie traitera de la programmation par composants et des raisons de son adoption par un grand nombre d'environnements de simulation scientifique. Nous essaierons de mettre en exergue les fonctionnalités attendues d'un modèle de composants par une application de pilotage, de visualisation ou de réalité virtuelle. Notre approche de la visualisation scientifique interactive ressemblant assez à cette dernière catégorie, nous visons à justifier, à l'issue de cette partie, l'introduction d'un nouveau modèle de composants qui lui soit plus approprié.

Chapitre 2

Programmation par composants

Sommaire

2.1 Définition

La programmation par composants -ou orientée composant (POC)- est une forme de programmation structurée [24] dont la finalité est d'améliorer la qualité (flexibilité, maintenabilité, fiabilité) de logiciels de plus en plus grands en en maintenant les coûts et durées de production [7]. En pratique, l'approche qu'elle adopte est de reproduire en génie logiciel le principe de composants réutilisables qu'on trouve en électronique et ce, par un ensemble de formalismes, de paradigmes de programmation et d'outils. Au final, une application *composite* (ou *modulaire*) exécutable sera un ensemble de tâches compilées, appelées *composants* ou *modules*, et communiquant entre elles selon un *protocole*. L'idée de composants logiciels fut énoncée pour la première fois en 1968 [68]. Son initiateur, Douglas McIlroy, soutint :

> *[...] software components [...] will offer families of routines for any given job [...] the purchaser of a component from a family will choose one tailored to his exact needs. He will consult a* catalogue *offering routines in varying degrees of precision, robustness, time-space performance, and generality [...] He will expect the routine to be intelligible, doubtless expressed in* a higher level language appropriate to the purpose of the component *[...] He will expect families of routines to be constructed on rational principles so that families* fit together as building blocks. *In short, he should be able safely to regard components as black boxes.*

Dans cette description, on peut relever les caractéristiques générales de la POC :

Interfaces Vu comme une boite noire, le composant cache ses algorithmes au reste de l'application. Pour communiquer avec les autres composants, il est équipé d'une couche extérieure, appelée *interface*. Les interfaces de tous les composants d'une application doivent

répondre à une même norme. Selon les cas, l'interface est caractérisée par des ports de services ou par des ports de données.

Réutilisation Les composants sont faits pour être partagés, entre les personnes et entre les applications. Cela tend à réduire le nombre de solutions personnelles pour un même problème et aide à converger vers des solutions communes plus robustes. L'approche modulaire permet également des mises à jour plus fréquentes et plus ciblées des logiciels complexes garantissant ainsi, pour l'utilisateur, plus de stabilité à son système sur le long terme.

Interchangeabilité Si le principe de réutilisation permet de préserver, aussi longtemps que possible, les parties valables d'un logiciel, c'est le principe d'interchangeabilité qui permet de mettre à jour celles qui ont besoin de l'être. L'interchangeabilité, obtenue grâce à une interface et une norme de description unifiées, facilite l'évolution des logiciels mais aussi la communication, l'expérimentation et la reproductibilité des résultats scientifiques.

Séparation des préoccupations *Separation of concerns*, en anglais. Il s'agit de faire en sorte que les composants assurent les fonctions les plus élémentaires possibles afin de ne pas avoir de fonctionnalités identiques lorsqu'ils se retrouvent dans la même application. C'est une condition à la réutilisation. On peut aussi parler de séparation des préoccupations de l'utilisateur de celles du développeur, l'interface dispensant le premier d'avoir connaissance de l'intérieur du composant pour savoir l'utiliser. C'est l'une des raisons pour lesquelles l'approche par composants est plébiscitée dans le domaine du calcul scientifique où l'utilisateur n'est pas toujours un informaticien aguerri. Enfin, il s'agit également d'une séparation des préoccupation entre les développeurs eux-mêmes puisque les différents composants d'une même application peuvent être développés en parallèle par les personnes qualifiées.

2.2 Modèles de composants

Un modèle de composants est une formalisation d'une structure de composant dotée d'une interface qui expose ou demande au reste de l'application certaines données ou fonctionnalités. Cette interface est constituée de *ports* qui peuvent être :

- D'entrée : utilise une donnée ou une fonctionnalité produite par un autre composant ;
- De sortie : fournit une donnée ou une fonctionnalité aux autres composants.

Dans l'implémentation d'un composant, on trouve deux types de code :

- le code fonctionnel : aussi appelé *code métier*, c'est l'implémentation des fonctionnalités du composant (calcul scientifique, filtrage, transformations géométriques, etc.) ;
- le code non fonctionnel : aussi appelé *code de gestion*, est celui faisant fonctionner le composant à l'intérieur du modèle : cycle de vie, communication, etc.

Outre le composant, un modèle définit également la connexion et l'application composite à savoir l'assemblage d'instances de composants par des connexions client-serveur. La composition est effectuée par l'utilisateur de l'application avant son lancement (composition *statique*) grâce à un langage de description d'architecture (*Architecture Description Language* ou *ADL*, en anglais). Certains modèles prévoient également que la composition puisse être modifiée pendant l'exécution (composition *dynamique*).

2.2.1 Modèles à composition spatiale

L'un des premiers modèles de composants fut présenté à la fin des années 1980 sous la forme d'un nouveau langage de programmation, l'Objective-C [18]. Si l'on se réfère à la finalité originelle de la programmation par composants qui est la séparation des préoccupations, le travail de composition d'une application revient à un simple regroupement de fonctionnalités dans un même espace virtuel. On parle, dans ce cas, de *composition spatiale* [11]. Au sein d'une application composite, les composants jouent alternativement les rôles de clients ou de serveurs les uns vis à vis des autres selon qu'ils invoquent une fonction chez un autre composant ou qu'ils en fournissent une. Un tel schéma de communication se nomme *Appel de procédure à distante* (*Remote Procedure Call* ou *RPC*, en anglais) [10]. On utilise également la dénomination *Invocation de méthode à distance* (*Remote Method Invocation* ou *RMI*, en anglais) lorsque le serveur est orienté objet. Le sens, la temporalité et le nombre des échanges entre composants seront dépendants de leurs codes fonctionnels. CORBA [26] et CCA [8], les deux grands représentants de ce type de modèles, sont employés dans des applications de simulation et de visualisation scientifique.

CORBA

Le Common Object Request Broker Architecture est un standard défini par le consortium ORB (Object Management Group) afin de permettre à des composants logiciels écrits dans différents langages et s'exécutant sur des machines différentes de communiquer ensemble et de ne former qu'une seule application. CORBA dispose d'un langage de définition d'interfaces (*Interface Definition Language* ou *IDL*, en anglais) abstrayant les types de données des codes fonctionnels et permettant de décrire les fonctionnalités que peut fournir chaque composant au reste de l'application. Un intergiciel appelé ORB (Object Request Broker) est en charge de la communication entre composants. Le style de programmation orienté-objet de CORBA évolua en composition en 2002 avec l'introduction du CCM (*CORBA Component Model*). Le composant CCM est constitué de :

- Une référence à l'instance de composant ;
- Des attributs configurables ;
- 4 types de ports :
  - la facette : port de sortie exposant une fonctionnalité du composant ;
  - le réceptacle : port d'entrée sollicitant une fonctionnalité externe ;
  - la source : port de sortie émettant ponctuellement des données à un ou plusieurs puits ;
  - le puits : port d'entrée d'une donnée ponctuelle émanant d'une ou de différentes sources.

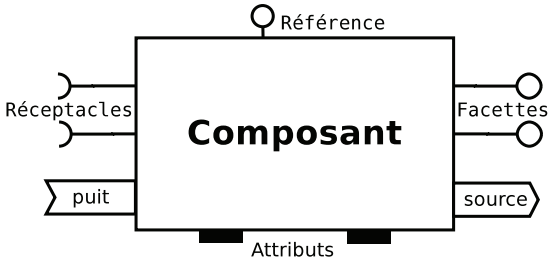


FIGURE 2.1 – Le composant CCM.

La figure 2.1 illustre un composant CCM. Les communications facette-receptacle sont synchrones : le client se met en attente jusqu'à réception d'une réponse ou levée d'une exception. Les communications source-puits sont, elles, asynchrones et à l'initiative du serveur.

CORBA est, aujourd'hui, un modèle très exhaustif qui définit de la structure du composant jusqu'à la procédure de déploiement automatique d'une application sur une architecture matérielle. Malgré cela et en particulier dans sa version CCM, il bénéficie d'un faible taux d'adoption au sein de la communauté scientifique en comparaison à CCA.

CCA

Common Component Architecture est un modèle de composants destiné au calcul haute performance et entretenu par un groupement de chercheurs et d'universitaires basés aux États Unis appelé le *CCA Forum*. Sa principale originalité est qu'il permet l'ajout et la suppression de composants à la volée. Un composant CCA peut être programmé en C, C++, Fortran, Java ou Python. Il peut être équipé de deux types de ports : client (*uses*) et serveur (*provides*). Ces ports ne participent pas à la définition d'un composant par l'utilisateur car ils peuvent être créés et supprimés pendant l'exécution. Au lieu de cela, sont déclarés, dans sa définition, les fonctions que le composant peut solliciter via ses ports client ou fournir via ses ports serveur. La figure 2.2 montre la connexion entre un composant A réclamant une fonctionnalité P et un autre composant B qui la fournit.

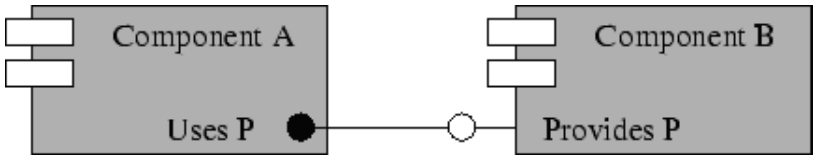


FIGURE 2.2 – Deux composants CCA connectés par leurs interfaces.

Le langage de description adopté par CCA est le langage SIDL (*Scientific Interface Définition Language*, en anglais) qui est riche de types de données spécifiques au calcul scientifique. À partir des descriptions SIDL des composants, un *proxy generator* ajoutera à leurs implémentations des instructions non fonctionnelles. Ces instructions appartiendront à l'une des catégories de services que définit CCA : communication, sécurité, création et gestion de processus, gestion de la mémoire et traitement d'erreurs. L'environnement CCA doit, lui aussi, fournir des services standard de ce genre. Enfin, CCA définit également des librairies pour le dialogue entre les composants et l'outil de composition (appelé *builder*) ainsi que pour gérer un répertoire de composants.

2.2.2 Flux de travaux : les modèles à composition temporelle

Un flux de travaux (*workflow*, en anglais) définit une chaîne de traitements. Les flux de travaux apparaissent dans toutes sortes de secteurs professionnels. Il peuvent décrire un procédé industriel, logistique, administratif voire un protocole scientifique expérimental. Cette forme d'organisation d'une activité améliore sa gestion (temps, ressources) et sa documentation (lisibilité, reproductibilité). L'origine du concept se situe probablement au début du XX^{e} siècle [83]. L'un des premiers usages du terme workflow en informatique remonte à 1985 dans le domaine de la gestion des processus métier (*Business Process Management* ou *BPM*, en anglais) [31]. L'aspect encore trop monolithique des premiers

outils différa, cependant, la démocratisation des flux de travaux informatiques à la décennie suivante où ils connurent un regain d'intérêt. En 1995, le consortium *Workflow Management Coalition* (WfMC) décrivit le flux de travaux informatique en ces termes [44] :

Définition 3 (Workflow) *Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal.*

L'objet du flux de travaux, selon cette définition, est d'automatiser un procédé métier (industriel, scientifique, etc.), réel ou virtuel, grâce à un contrôle informatique. Ce procédé est présenté comme une chaîne de traitements dans laquelle des objets ou des informations sont passés d'une tâche de la chaîne à une autre lorsqu'une condition est remplie. L'exemple le plus évident de condition de passage à la tâche suivante est l'accomplissement de la tâche courante. Un flux de travaux peut donc être vu comme une application composite, chaque tâche pouvant y être représentée par un composant, augmentée d'une logique temporelle. On peut alors parler de *composition temporelle* [11]. En outre, dans ce type de modèles, la séparation des composants est plus prononcée que dans ceux à composition spatiale puisque c'est la transmission de données qui tient lieu de communication et non plus une invocation de méthode, forcément plus intrusive.

Les flux de travaux adoptent généralement l'un des deux modes d'exécution suivants :

- Flot de données (*dataflow*, en anglais) : l'exécution d'un composant est conditionnée par la réception de paquets de données, souvent appelés *jetons* (*tokens*, en anglais) ou *messages*, sur ses ports d'entrée ;
- Flot de contrôle (*control flow*, en anglais) : le modèle dispose d'objets spécifiques pour contrôler l'exécution des composants voire d'instaurer des boucles et des conditionnelles au sein du flux.

Plus nombreux et moins généraux que les modèles à composition spatiale, les modèles de flux de travaux sont généralement associés à des logiciels de gestion de flux de travaux (*Workflow Management Systems* ou *WfMS*, en anglais). Le WfMC définit ces logiciels ainsi :

Définition 4 (Workflow Management System) *(A Workflow Management System is) A system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic.*

D'après cette définition, un WfMS permet donc de concevoir et d'exécuter des flux de travaux à partir d'un logiciel et ce, automatiquement selon un ordre connu par l'ordinateur. Un WfMS peut piloter un procédé industriel réel. Il peut aussi conduire un procédé complètement informatique comme dans le cadre d'une simulation scientifique. C'est précisément ce second type d'applications qu'il nous intéressera d'étudier au prochain chapitre. On connaît, dans ce cadre et depuis 1999, le terme *e-science* pour désigner les applications scientifiques de calcul intensif pouvant nécessiter des ressources matérielles distribuées. Les applications de visualisation et de réalité virtuelle à des fins de recherche scientifique rentrent dans cette catégorie. De la même manière, on désigne par l'expression *système de gestion de flux de travaux scientifiques* (*Scientific Workflow Management Systems* ou *SWfMS*, en anglais) [58] les systèmes permettant aux chercheurs de modéliser, d'organiser et d'exécuter, sous la forme de flux de travaux, des expériences virtuelles.

2.3 Conclusion

Les propriétés de l'approche par composants énoncées à la section 2.1 sont particulièrement avantageuses pour le monde de la recherche, familier des projets complexes, longs et collaboratifs. En premier lieu, le support de plusieurs langages de programmation, à commencer par les langages dominants dans le calcul scientifique (C, C++ et Fortran) a pour effet d'allonger la durée de vie des composants. Leur abstraction par une interface de description permet à un scientifique de récupérer un programme réalisé par un collègue et, sans avoir à en comprendre l'implémentation, de l'intégrer à son propre environnement de simulation afin de le tester et de le comparer à sa propre solution. La réutilisation, ici facilitée, promeut la diffusion scientifique et la fiabilité des résultats et ce, en décuplant les jeux de test. Dans le cadre d'un travail collaboratif, la décomposition d'une application complexe permet aussi à ses différentes parties d'être développées en parallèle par les personnes qualifiées. Les applications de visualisation scientifique multiphysiques sont typiquement susceptibles de tirer parti de ce type d'approches.

La principale faiblesse des modèles à composition spatiale vis a vis des applications scientifiques est l'absence de logique temporelle ou, du moins, son enfouissement dans l'implémentation des composants. L'emploi d'un des schémas de communication disponibles (synchrone, asynchrone ou autre) nécessite clairement une connaissance des comportements internes des composants et donc la sollicitation de leurs développeurs. Cela laisse penser que la séparation des préoccupations entre personnes n'est pas totalement atteinte avec ce genre de modèles.

Les modèles à composition temporelle pallient à ce problème. De plus, la composition temporelle se trouve plus proche de la programmation classique que ne l'est la composition spatiale en ce qu'elle permet non seulement d'enchaîner les composants comme des morceaux de code au sein d'un programme mais aussi d'explicitement établir des boucles ou des portions parallèles. Au prochain chapitre, nous étudions différents modèles concrets spatiaux ou temporels ainsi que leurs méthodes de composition associées.

Chapitre 3

Composition d'applications scientifiques

Sommaire

3.1 L'environnement de composition d'applications scientifiques

L'expression anglaise *scientific workflow* est, par abus de langage, parfois employée pour désigner une application scientifique composite en général. Aussi peut-on trouver des systèmes implémentant un modèle spatial et se déclarant SWfMS [32]. Pour éviter toute ambiguïté, nous préférons adopter l'expression plus générale d'*environnements de composition d'applications scientifiques* pour désigner les systèmes permettant de composer et d'exécuter des applications scientifiques. Pour plus de commodité, nous nous y référerons parfois par les simples termes de *systèmes* ou *environnements*.

Dans cette section, nous énumérons les principaux points à considérer dans l'étude d'un modèle de composition d'applications scientifiques. Le point de vue relativement théorique de cette étude nous fera volontairement éluder les aspects d'implémentation, de distribution et d'ordonnancement malgré qu'ils soient souvent pris en charge par les environnements.

3.1.1 Représentation

Une application est généralement illustrée par un graphe orienté dont les sommets sont les composants et les arêtes représentent le transfert d'activité (et non nécessairement de

données comme l'explique la section 3.1.2) entre composants. Si le modèle intègre la notion de composants parallèles, il peut offrir deux niveaux de représentation de l'application : le graphe plié d'abord et, accessoirement, le graphe déplié. Dans un graphe plié, un composant, même parallèle, n'est représenté que par une seule instance de composant tandis qu'un graphe déplié met en évidence les différentes instances de ses composants parallèles. Une autre raison de représenter l'application à plusieurs niveaux de détail différents est la présence, dans certains modèles, de composants hiérarchiques, à savoir la possibilité d'encapsuler une composition dans un composant pour la réutiliser comme sous-partie d'une autre composition.

Certains modèles autorisent les graphes cycliques (*non Directed Acyclic Graph* ou *non-DAG*, en anglais) [92]. D'autres, par exemple pour éviter les interblocages, ne permettent que les graphes acycliques (*Directed Acyclic Graph* ou *DAG*, en anglais). Enfin, la plupart des modèles développent leurs propres représentations adaptées aux domaines d'application qu'ils visent mais les réseaux de Petri -éventuellement colorés- [47] et les diagrammes d'activité UML [72] suffisent, parfois, à illustrer certains [70,93].

3.1.2 Logique

Le modèle peut adopter un mode de composition spatial, temporel flot de données ou temporel flot de contrôle. Une approche exclusivement flot de données aboutit à une exécution séquentielle du flux de travaux ce qui pourrait ne pas suffire pour modéliser certaines applications complexes. Même si, dans l'absolu, les principaux schémas de contrôle (boucles, conditionnelles) peuvent être implémentés dans des composants, plusieurs environnements adoptent aujourd'hui un modèle hybride en offrant des objets de contrôle prédéfinis afin de faciliter la tâche de composition à l'utilisateur. Ces objets se matérialisent sur le graphe sous la forme de sommets et d'arêtes supplémentaires.

3.1.3 Cycle de vie du composant

Cette notion importante intervient dans la distinction entre modèles à composition temporelle et modèles à composition spatiale. Les modèles pour lesquels la coordination de l'application joue un rôle important prennent en compte le comportement interne de leurs composants et, parfois même, ils le définissent. Ce comportement peut inclure :

- l'instant de démarrage du composant : le composant peut démarrer en même temps que l'application, lorsqu'il reçoit des données, lorsqu'il est déclenché par un autre composant ou par l'utilisateur, etc ;
- le caractère bloquant ou non bloquant des ports : un composant peut-il s'exécuter alors que tous ses ports d'entrée n'ont pas été alimentés ?
- la durée de vie du composant : l'opération exécutée par le composant peut être de nature semelfactive ou itérative.

3.1.4 Couplage

La recherche scientifique produit une grande quantité de codes source que la programmation par composants doit permettre de réutiliser. Il s'agit, ici, d'évaluer les moyens offerts par l'environnement pour créer des composants et, surtout, son aptitude à intégrer, en tant que composants, ces codes dits *patrimoniaux* (*legacy code*, en anglais). Bien que cela ne soit pas une propriété du modèle sur le plan formel, les possibilités de couplage peuvent être conditionnées par le cycle de vie du composant. Différentes solutions, qui ne se valent pas en terme de performances, peuvent être proposées :

L'annotation L'environnement peut fournir une ou plusieurs librairies de fonctions à insérer dans le code source afin qu'il puisse communiquer avec les autres composants. C'est une solution qui nécessite d'avoir accès au code source du programme ce qui n'est pas toujours possible.

La passerelle L'utilisateur doit programmer un composant -dont un squelette peut être fourni par l'environnement- en charge de communiquer avec le programme externe selon un protocole de communication inter-processus [82] : lecture/écriture dans un fichier ou dans un segment de mémoire partagée, sockets, etc. Le manuel d'utilisation du programme peut renseigner sur les protocoles de communication qu'il utilise.

L'enveloppement (*wrapping*, en anglais) Le programme peut être instancié en tant que sous processus d'un composant et d'éventuels paramètres initiaux peuvent lui être communiqués dans sa ligne de commande de lancement. Le composant est alors la vitrine du programme du point de vue de l'application composite. L'enveloppement est une solution pour l'exécution du programme et non pour la communication avec lui. Pour cela, il est souvent utilisé de pair avec une des deux précédentes techniques.

3.1.5 Composition

La composition d'une application peut, selon les systèmes, se faire de deux manières :
- Textuelle : l'utilisateur saisit dans un langage dédié -souvent de script ou bien proche du XML- les composants de son applications et leur connexions ;
- Graphique : Le système offre une interface graphique et l'utilisateur peut assembler son application par glisser-déposer.

Dans un cas comme dans l'autre, les systèmes tendent, aujourd'hui, à de plus en plus assister l'utilisateur dans cette étape. Cela se matérialise essentiellement par une vérification, sur la base du typage des ports, de la compatibilité des ports connectés.

En particulier s'il est destiné à des non informaticiens, le système peut masquer certaines technicités de son modèle de composants derrière un modèle de composition -celui auquel a accès l'utilisateur- plus simple et accompagné d'un plan de transformation automatique vers le modèle de composants [38, 71, 81]. On pourra alors aussi juger le système sur le degré d'abstraction que fournit son modèle de composition par rapport à son modèle de composants.

3.2 Exemples d'environnements

Depuis une quinzaine d'années, se sont développés un grand nombre d'environnements de composition, en particulier pour la gestion de flux de travaux scientifiques [22], et presque autant de modèles et d'approches différentes pour composer et coordonner des applications scientifiques haute performance. Sur la base des propriétés majeures énumérées à la section précédente, nous nous attacherons à comparer différents modèles et méthodes de composition à travers certains de leurs représentants dans le domaine du calcul et de la visualisation scientifique. Notre objectif est d'évaluer, à travers leurs usages, leur adéquation pour une application de visualisation scientifique interactive et, éventuellement, de déduire les propriétés qui caractériseraient un modèle plus approprié.

3.2.1 Modèles à composition spatiale

Ces modèles sont représentés dans notre étude par les systèmes EPSN [29], TENT [32], SCIJump (anciennement SCIRun 2) [73] et ICENI [65]. Les deux premiers se basent sur CORBA, le troisième sur CCA et le dernier définit son propre modèle. EPSN est employé dans plusieurs disciplines scientifiques comme la mécanique des fluides, la météorologie et la dynamique moléculaire. TENT est sollicité pour des applications de mécanique des fluides, de mécanique structurelle et de thermodynamique. Basé sur SCIRun [88], SCIJump est valable pour les même types d'applications que son prédécesseur comme le bioélectromagnétisme. Enfin, ICENI a été utilisé pour réaliser des simulations climatologiques.

Modèle

EPSN et TENT utilisent l'infrastructure de communication de CORBA sans toutefois reposer sur le modèle CCM. Pour communiquer avec l'ORB, chaque processus de l'application EPSN se voit attacher un processus léger (*thread*) relais encapsulant un serveur CORBA capable d'y accéder en lecture/écriture de manière asynchrone. Une approche semblable est employée par TENT qui enveloppe, dans un composant, chaque élément fonctionnel de l'application. Ces composants sont en charge de contrôler l'exécution du programme enveloppé et de surveiller ou de piloter certains paramètres. Ils communiquent entre eux par *événements* (*events*, en anglais).

Le composant ICENI est, lui, défini par les informations suivantes :
- Un nom ;
- Un ensemble de ports d'entrée ;
- Un ensemble de ports de sortie ;
- Éventuellement des propriétés.

L'application ICENI se présente sous la forme d'un graphe non-DAG de composants connectés via leurs ports. Les connexions multiples y sont autorisées.

Afin de favoriser l'échange de composants avec d'autres environnement de composition, SCIJump se présente, lui, comme un modèle de méta-composants soit un environnement capable d'interfacer plusieurs intergiciels et de les faire fonctionner comme une seule et même application. Parmi ces intergiciels figurent CCA, CORBA, SCIRun (voir l'architecture de ce dernier à la section 3.2.2). SCIJump porte, ici, le principe de réutilisation au delà des limites du modèle.

La notion de composant hiérarchique est présente dans EPSN, TENT et SCIJump et absente de ICENI.

Composition

L'application EPSN typique est composée d'une simulation -un calcul ou plusieurs calculs couplés- parallèle jouant le rôle de serveur et entourée de plusieurs clients de visualisation/pilotage qui peuvent s'y connecter à la volée. La programmation d'une application EPSN passe par une annotation du code de simulation C, C++ ou Fortran pour, notamment, délimiter la boucle de simulation ou une zone autorisée d'accès aux données. Cette annotation doit être accompagnée d'un fichier XML synthétisant la structure du programme et recensant les variables partagées avec le système ainsi que les types d'interaction possibles.

La composition d'applications TENT, elle, se fait par le biais d'une interface graphique par glisser-déposer. Toutes les options de couplage évoquées à la section 3.1.4 peuvent être envisagées pour l'intégration de codes C, C++, Fortran ou C-Python. Une classe C++

générique est fournie pour l'écriture du composant enveloppant. Ce dernier doit également être décrit au reste de l'application par une IDL CORBA spécifiant les événements qu'il est susceptible d'envoyer ou de recevoir.

Dans ICENI, la composition peut se faire de manière textuelle en XML ou graphique grâce à un plugin pour l'environnement de développement intégré NetBeans [75]. Les composants doivent être créés à partir de trois fichiers XML de description : un pour la structure, un pour le comportement et un pour l'implémentation. Le premier contient globalement les éléments donnés dans la description ci-dessus du modèle. Le second permet de préciser, au niveau de chaque port, si la communication est une invocation de méthode ou une transmission de données. Il est également possible d'attacher à chaque port d'entrée un flot de contrôle d'instructions à exécuter lorsque ce port est activé. Le dernier fichier, quant à lui, ajoute des informations propres à l'implémentation telles que les types Java des données échangées par les ports ou d'éventuels chemins d'accès pour des dépendances de fichiers. À partir de ces trois fichiers, ICENI, génère un squelette Java des sources du composant que l'utilisateur doit compléter. Outre cette méthode, ICENI ne fournit pas de solutions facilitant l'intégration de codes existants.

Enfin, concernant SCIJump, la composition d'une application se fait via une interface graphique à partir de composants développés dans leurs environnements d'origine. Dans l'absolu, l'utilisateur est censé pouvoir intégrer n'importe quel modèle de composants à condition de fournir un fichier (en langage eRuby) décrivant les équivalences avec l'autre modèle avec lequel il souhaite l'interfacer. En se basant sur ces traductions et sur les descriptions IDL des composants, le compilateur SCIJump est capable de générer automatiquement les composants intermédiaires entre deux composants appartenant à des modèles différents. L'efficacité de ces relais du point de vue de la performance est toutefois discuté par ses auteurs.

SCIJump propose trois niveaux de synchronisation pour une communication :

- Synchrone : le client doit attendre une réponse du serveur avant de continuer ;
- Asynchrone non bloquante : le client peut reprendre son activité après une invocation et vérifier périodiquement la réception d'une réponse ;
- Asynchrone à sens unique : il s'agit de l'invocation d'une procédure, aucune réponse de la part du serveur n'est attendue.

Le pilotage et la visualisation font partie des objectifs des 4 systèmes. ICENI et EPSN fournissent des clients génériques pour le pilotage et la visualisation. EPSN dispose également d'une librairie de couplage avec AVS/Express [30], VTK [45] et ParaView [42].

Fonctionnement

Les applications générées par ces différents environnements sont destinées à s'exécuter de manière autonome, directement à partir des interfaces graphiques de composition lorsqu'il y en a. L'application EPSN, par exemple, suit le plan d'exécution fourni dans la description XML de la simulation. Dans ICENI, un flux de travaux global est synthétisé à partir des fichiers décrivant les comportements des composants. Nous n'avons pas, pour autant, qualifié la composition dans ICENI de temporelle car ce flux de travaux est déduit par le système pour optimiser l'ordonnancement et non pas réalisé consciemment par l'utilisateur. L'application TENT, elle, fonctionne selon le modèle d'événements JavaBeans [27]. Enfin, l'ordre d'exécution des composants SCIJump est dicté par les politiques de synchronisation choisies au niveau local. SCIJump gère aussi de manière exhaustive et transparente la communication entre composants parallèles (via MPI [16] ou tout système similaire), y compris de nombres d'instances différents (communication N x M).

L'hétérogénéité des modèles et environnements de développement laissant supposer une variété de ressources matérielles, l'exécution de l'application SCIJump a été spécifiquement pensée pour la grille avec, notamment, une répartition circulaire des données et la possibilité d'un pilotage collaboratif. Dans EPSN, les requêtes de l'utilisateur peuvent concerner les données (lecture/écriture de variables) ou l'exécution (pause, play). L'environnement EPSN accorde une importance particulière à la cohérence des différents processus parallèles. Il répond à ce problème par un système d'indexation spatio(position dans le code)-temporel(numéro de l'itération, si jamais la ligne se trouve dans une boucle) des points d'instrumentation dans le code [28]. Cet index est appelé *date*. À chaque requête de l'utilisateur, le système planifie le changement à la même date dans tous les processus après s'être assuré qu'aucun ne l'a déjà dépassée. Ce mécanisme a l'avantage d'assurer une cohérence en écriture des processus sans les synchroniser. Ceci n'est pas vrai en lecture où le système est susceptible de mettre en attente certains processus. Par ailleurs, un système de date globale [80] permet la cohérence de plusieurs codes de calculs différents au sein de la même application ce qui accroît l'intérêt de cette approche.

Notons, enfin, dans TENT [49], une fonctionnalité très répandue dans les environnements de composition d'applications scientifiques qui consiste à tracer chaque exécution de l'application et de stocker cette trace dans une base de données à des fins d'analyse ou de débogage. Cet historique, qui peut comprendre l'ordre d'exécution des composants ou bien la date, le contenu et le canal emprunté par chaque donnée, est appelé *provenance* de l'application.

3.2.2 Modèles à composition temporelle flot de données

Cette approche est représentée par deux systèmes contemporains l'un de l'autre et aux modèles relativement similaires : SCIRun [88] et CoViSE [90]. Tous deux sont des environnements pour la simulation et la visualisation de problèmes scientifiques.

Modèle

SCIRun adopte une architecture flot de données élémentaire. Dans le vocabulaire de SCIRun, une application est représentée par un *réseau* (*network*, en anglais) non-DAG de *modules* reliés par des *connections*. Les modules sont caractérisés par :

- Un nom ;
- Zéro, un ou plusieurs ports d'entrée typés de données ;
- Zéro, un ou plusieurs ports de sortie typés de données ;
- Des paramètres modifiables à la volée via une fenêtre de pilotage dédiée au module.

Le module CoViSE est similaire au module SCIRun et ne s'en distingue que par deux propriétés supplémentaires :

- Un attribut désignant sa fonction (entrée/sortie, simulation, filtre, rendu, etc.) ;
- La possibilité de rendre un port de sortie dépendant d'un port d'entrée auquel cas si le premier est connecté, le second doit l'être également.

Le modèle SCIRun accepte les cycles dans le réseau et intègre la notion de composants hiérarchiques sous le nom de *sous-réseau* (*sub-network*, en anglais). Les ports du sous-réseau sont les ports non connectés des modules qui se trouvent à l'intérieur. Ces deux fonctionnalités sont absentes du modèle CoViSE.

Composition

Dans SCIRun et CoViSE, la composition d'un réseau se fait entièrement via interface graphique. Les deux systèmes fournissent quelques modules spécialisés. Pour SCIRun, il s'agit, par exemple, de modules pour le Bioélectromagnétisme, son domaine de prédilection, ainsi que quelques composants d'interfaçage avec des programmes externes C, FORTRAN ou Matlab. Dans CoViSE, le couplage de modules avec des simulations externes C, C++ ou Fortran nécessite leur annotation avec une librairie dédiée. Autrement, l'utilisateur peut programmer ses propres composants en C++. De plus, dans SCIRun, il peut créer leurs interfaces graphiques en Tcl/Tk pour les piloter. Dans CoViSE, ces interfaces graphiques sont générées automatiquement au moment de la composition.

Fonctionnement

Le fonctionnement des réseaux SCIRun et CoViSE est séquentiel et synchrone. Dans SCIRun, lorsque l'utilisateur commande l'exécution d'un module, un programme interne, appelé *scheduler*, planifie celles des modules en amont (pour lui fournir des données) et en aval (pour utiliser ses données) de celui-ci. Toutefois, l'implémentation du module laisse la possibilité d'envoyer plusieurs résultats (intermédiaires) en une itération. Dans ce cas, le scheduler est capable de programmer une exécution itérative des autres modules qui en dépendent. Les modules CoViSE, eux, possèdent un paramètre pour indiquer s'ils initient la chaîne d'exécution ou pas. Un module peut également être lancé individuellement ce qui entraînera l'exécution, par la suite, de toute la partie du réseau qui en dépend.

Le pilotage des modules, dans un système comme dans l'autre, se fait uniquement via des boîtes de dialogue. En ce sens, malgré sa compatibilité avec un grand nombre de périphériques d'interaction et avec des systèmes d'affichage immersifs à travers le module OpenCOVER, CoViSE fournit une réalité virtuelle limitée dans la mesure où l'interaction s'y restreint à de la navigation.

3.2.3 Modèles à composition temporelle flot de contrôle

Les modèles purement flot de contrôle sont beaucoup plus rares que les modèles flot de données. La raison essentielle que nous voyons à cette absence est l'originalité même d'un flux de travaux n'explicitant pas le passage de données d'une tâche à l'autre. Cactus [39], un environnement pour la composition d'applications scientifiques haute-performance, fait partie de cette catégorie. La communication entre ses composants, appelés *épines* (*thorns*, en anglais), se fait par invocations de fonctions comme dans CORBA ou CCA. Ces invocations ne sont cependant pas arbitraires mais sont planifiées par l'utilisateur dans un fichier de programme (*schedule*, en anglais). En ce sens, le modèle Cactus correspond bien à un modèle de flot de contrôle.

Modèle

Le modèle Cactus est un modèle particulièrement concret. L'application, telle que décrite par l'utilisateur n'est pas réinterprétée mais directement compilée pour les architectures cibles sous le nom de *configuration*. Ce modèle emprunte, par ailleurs, à la fois aux modèles de composition spatiale et à la programmation orientée objet (POO). Les épines, par exemple, n'ont pas de ports et s'apparentent à des classes. Elles sont définies par :

- D'un point de vue POO, les éventuelles parentés avec d'autres épines : héritage, amitié, etc ;
- Les fonctions fournies ou utilisée par l'épine ;

- L'ensemble des variables provenant du code de l'épine et partagées avec le reste de l'application.
- Un ensemble de paramètres et leurs propriétés : nom, type, visibilité, modifiablilité pendant l'exécution, etc ;
- Un plan d'exécution donnant l'ordre d'appel des fonctions de l'épine les unes par rapport aux autres, par rapport à des étapes clés de l'application (démarrage, initialisation, évolution, fin, etc.) ou selon la valeur d'un paramètre ;

Pour compléter le modèle, un élément intermédiaire central appelé la *chair* (*flesh*, en anglais), transparent pour l'utilisateur, joue le rôle de moteur coordinateur de l'application.

Composition

Cactus fournit quelques dizaines d'épines réalisant des opérations élementaires (lecture/écriture dans un fichier, communication réseau, calculs, etc) et une épine générique pour la visualisation 3D appelée IsoView. La programmation d'épines et la composition d'applications se font de manière entièrement textuelle. La spécification d'une épine par les attributs énumérés ci-dessus se répartit entre 4 fichiers distincts écrits dans un langage de script appelé le *Cactus Configuration Language* (CCL). L'implémentation du composant, qui peut être en C, C++ ou Fortran, doit inclure quelques lignes d'annotation en CCL pour indiquer les variables, paramètres et fonctions exposées. Cependant, la logique du composant (conditions, boucles, parallélisme) doit être exprimée dans le fichier de plan d'exécution et non dans le langage cible. Cela identifie les épines à des codes élémentaires programmés pour le modèle plutôt qu'à de larges programmes de calcul transformés par l'utilisateur.

La composition d'une configuration, entièrement textuelle, se résume ensuite à lister les épines impliquées et, éventuellement, à initialiser certains paramètres. Avant le lancement, la chair vérifie la consistance des plans d'exécution.

Fonctionnement

Afin de faciliter la programmation d'applications scientifiques, souvent basées sur un processus itératif, la chair adopte un mode de fonctionnement cyclique. Elle découpe l'application en étapes clés auxquelles l'utilisateur peut se référer dans les plans d'exécution de ses composants :

1. Démarrage (*Startup*) ;
2. Initialisation (*Initial*) ;
3. Portion itérative :
   (a) Pré-itération (*Prestep*) ;
   (b) Itération (*Evol*) ;
   (c) Post-itération (*Poststep*) ;
   (d) Analyse des résultats (*Analysis*) ;
   (e) Affichage des résultats (*Output*) ;
4. Fin (*Terminate*) ;

La visualisation interactive des résultats est possible mais n'est pas courante dans Cactus. En dehors de l'emploi d'IsoView, l'utilisateur aura le choix entre programmer sa propre épine de visualisation ou communiquer avec un visualiseur externe par sockets ou par fichier intermédiaire. Le pilotage, soit la modification de paramètres à la volée, est permis par l'infrastructure, là encore à condition de développer ses propres interfaces.

3.2.4 Modèles à composition temporelle hybride

Les modèles hybrides sont, généralement, des modèles flot de données enrichis de fonctionnalités flot de contrôle. Historiquement, la vision du flux de travaux comme un graphe de flot de données domine l'état de l'art. Cela est notamment dû au caractère séquentiel des premiers flux représentant des chaînes de traitement. L'élargissement des flux de travaux à toute une variété d'utilisations a fait naître le besoin d'une coordination plus précise. Un usage typique est un composant qui doit en déclencher un autre auquel il n'envoie pas de données. Dès lors, des fonctionnalités flot de contrôle ont été greffées aux modèles tout en veillant à ce qu'elles ne soient ni envahissantes ni indispensables afin de préserver une ligne d'utilisation claire pour l'usager.

Nous illustrerons ce sujet à travers Triana [14] et Kepler [60], deux systèmes aux modèles flot de données relativement semblables mais qui n'accordent pas la même importance à leurs modèles flot de contrôle.

Modèle

Les composants de Triana et de Kepler, appelés respectivement *tâches* (*tasks*, en anglais) et *acteurs* (*actors*, en anglais), ont des structures similaires. Ils sont définis par :

- Un nom ;
- Des ports typés d'entrée et de sortie de données ;
- Des paramètres.

Les deux modèles intègrent la notion de composants hiérarchiques sous le nom de *groupe de tâches* dans Triana et d'*acteurs composites* dans Kepler. Ils se distinguent, cependant, par des types de ports spécifiques. Dans Triana, des ports d'entrée de déclenchement (*trigger input node*, en anglais) permettent à une tâche d'être activée par une autre sans en recevoir de données. Kepler, lui, définit les types de ports non exclusifs suivants :

- Ports d'entrée-sortie : servent à la fois à la réception et à l'envoi de données ;
- Ports d'entrée multiples : peuvent être connectés à plusieurs ports de sortie ;
- Ports-paramètres : chacun est relié à un paramètre particulier du composant et le met à jour avec les valeurs qu'il reçoit ;
- Ports externes : ce sont les ports de l'acteur composite, à relier aux ports des composants qu'il contient.

Les deux modèles ne distinguent pas composants et structures de contrôle. Conditionnelles et boucles sont ainsi dirigées par des composants préprogrammés et l'utilisateur a le loisir de programmer ses propres composants de contrôle. De plus, dans Kepler, le flot de contrôle de l'application est dirigé par un élément particulier appelé *réalisateur* (*director*, en anglais). Kepler reprend, ici, la métaphore empruntée au cinéma par son prédécesseur Ptolemy II [25] selon laquelle les acteurs "agissent" et le réalisateur "coordonne". Le réalisateur, unique dans chaque application, définit notamment le degré de synchronicité entre les acteurs ou encore le nombre de fois que doit s'exécuter chaque acteur ou tout le graphe. L'environnement compte les types de réalisateurs suivants :

- Synchronous Dataflow (SDF) : exécution séquentielle, synchrone et déterministe du graphe dépourvu donc de conditionnelles ;
- Dynamic Dataflow (DDF) : exécution synchrone mais non déterministe du graphe ;
- Process Networks (PN) : les acteurs fonctionnent de manière asynchrone et leurs produits sont stockés dans des buffers en attendant d'être consommés ;
- Discrete Events (DE) : la circulation des jetons est orchestrée par une horloge globale en fonction des estampilles temporelles qu'ils contiennent ;

– Continuous Time (CT) : destiné au calcul de systèmes dynamiques évoluant dans le temps ;

Composition

Les composants dans Triana et Kepler sont supposés être particulièrement élémentaires. Il s'agit, en effet, de classes Java dont la méthode principale est identifiée par le nom *process()* dans Triana et *prefire()* dans Kepler. L'implémentation d'une tâche doit être accompagnée d'une description XML renseignant son nom, ses ports et ses paramètres. Actuellement, chacun des deux environnements fournit quelques centaines de composants destinés aux besoins de sa communauté d'utilisateurs. Il s'agit, globalement, de composants pour le traitement du signal, du son, de l'image, du texte et de visualisation.

Dans les deux environnements, la composition d'une application se fait graphiquement par glisser-déposer des composants dans la zone de travail puis leur raccordement. Une vérification de compatibilité entre les ports est alors automatiquement effectuée. L'utilisateur de Kepler doit aussi choisir un réalisateur en fonction de la nature de son application. Pour lui faciliter la tâche, le manuel d'utilisation de Kepler contient un questionnaire sur l'application permettant de cibler le réalisateur le plus adapté. Nous l'avons résumé dans le tableau 3.1.

	DE	CT	PN	SDF	DDF
Dépend du temps	X	X			
Contient du calcul différentiel		X			
Contient des processus indépendants ou distribués			X		
Est une simple transformation de données				X	

TABLE 3.1 – Critères de choix du réalisateur pour un flux de travaux Kepler

Le couplage d'une application Triana avec un programme externe est possible par enveloppement et usage d'un composant passerelle. Quelques expérimentations ont été publiées, notamment avec Cactus dans le rôle du programme externe [40,87]. Kepler, lui, fournit un acteur standard d'enveloppement nommé *ExternalExecution*. Cet acteur peut également jouer le rôle de passerelle en alimentant en chaînes de caractères un programme enveloppé communiquant par lignes de commande. Avec ces solutions de couplage, la récupération des résultats du programme externe a toutefois toujours lieu par lecture de fichiers partagés ce qui limite l'interaction, pendant l'exécution, à un caractère ponctuel.

Fonctionnement

L'exécution d'un flux de travaux dans ces systèmes se fait depuis l'interface graphique qui sert à la composition. Les composants n'ayant pas de connexions entrantes initient le flux. Les autres s'exécutent ensuite dès que tous leurs ports bloquants (qui empêchent l'exécution du composant tant qu'ils ne sont pas alimenté) sont servis. Le comportement global de l'application Triana est, de manière générale, tributaire de l'implémentation des composants sur laquelle le modèle n'impose rien. Cela concerne, par exemple, le caractère bloquant ou non bloquant des ports d'entrée. Les applications Kepler, elles, s'exécutent selon la politique de leurs réalisateurs. La restriction à un seul réalisateur par application assure une certaine cohérence à l'ensemble mais limite la complexité des applications qu'il est possible de modéliser. Il est également possible d'avoir, à l'intérieur de chaque acteur

composite, un réalisateur différent de celui de l'application. Ceci requiert une bonne maîtrise du modèle afin d'éviter les incohérences. Enfin, les deux modèles de flot de données sont non-DAG et sont donc vulnérables aux interblocages. Typiquement, un interblocage survient lorsqu'il existe, dans le graphe, un cycle ne passant que par des ports bloquants. Indirectement, chaque composant se retrouve alors à attendre son propre résultat avant de démarrer. Pour débloquer ces situations, Kepler propose d'intercaler dans le cycle un acteur nommé *SampleDelay* capable d'émettre un jeton d'une valeur fixe au démarrage de l'application.

Triana et Kepler sont, fondamentalement, des systèmes de *traitement par lot* (*batch processing*, en anglais) et non de visualisation ni de pilotage de simulations. L'usage de la visualisation s'y restreint, généralement, au monitoring ou au post-processing [41,84]. Par ailleurs, en terme de post-processing, Kepler est, au même titre que TENT, capable de gérer la provenance des données [19].

3.3 Conclusion

Loin d'être exhaustif sur les systèmes de composition d'applications scientifiques actuellement employés, l'objet de ce chapitre était de recenser un maximum des propriétés que l'on peut y trouver. Au sujet des modèles à composition spatiale, il est clair que les IDL permettent une séparation des préoccupations entre développeur et utilisateur mais on note que certaines utilisations propres à la composition demandent une connaissance approfondie du code fonctionnel. On peut ici prendre pour exemple la cohérence de données issues de plusieurs processus d'une application EPSN. L'approche présentée prouve l'importance de cette cohérence dans une application scientifique hétérogène mais aussi la difficulté de sa mise en œuvre. La bonne connaissance des algorithmes impliqués qu'elle impose ainsi que la description XML très précise qu'elle en demande la rendent peu applicable dans un contexte de partage/réutilisation de codes.

Le flux de travaux paraît plus adapté que la composition spatiale à la mise en place d'expériences scientifiques virtuelles à partir de codes patrimoniaux. Ces codes sont initialement conçus pour remplir des fonctions spécifiques de manière autonome : lecture d'un fichier, calcul, conversion de données, etc. Si l'application inclut, en plus, une visualisation 3D des résultats, elle pourra également comporter un ou plusieurs codes de visualisation et, éventuellement, de gestion de périphériques d'interaction. En tant que composant, chacun de ces codes remplira un rôle au sein de l'application modulaire : récupération des données d'entrée, pré-traitement, simulation, post-traitement, rendu graphique, affichage, etc. Situer dans le temps l'intervention de chacun facilite la compréhension du déroulement de l'application, la communication autour du projet et la localisation des erreurs. De plus, ces codes pouvant provenir de personnes différentes, un schéma flot de données semble d'autant plus judicieux qu'il occulte les routines internes du composant pour n'exposer à l'extérieur que les types de données qu'il échange.

Un modèle doit s'équiper d'un flot de contrôle afin d'être capable de représenter des schémas de coordination autres qu'une chaîne de traitements séquentielle. Ce chapitre a montré différentes façons d'intégrer ce flot de contrôle dans un modèle. Bien que celui Cactus ait un référentiel temporel unique, sa description dispersée entre les composants rend difficile, au moment de sa conception, la perception globale du déroulement de l'application. Dès lors, l'emploi d'objets de contrôle exogènes [52] est l'approche la mieux à même de séparer code fonctionnel et code non fonctionnel. Afin d'éviter toute ambiguïté entre ces *composants de contrôle* ou *connecteurs* et les composants conventionnels, il est alors préférable que le modèle décrive également le cycle de vie de ces derniers.

Au regard des définitions données au chapitre 1, nous constatons que les différents systèmes, même ceux dont elle est la finalité [88, 90], n'utilisent pas la visualisation comme vecteur d'interaction avec la simulation. Ils relèvent plutôt du pilotage qui consiste à changer manuellement quelques valeurs numériques et à observer le résultat. Dans la littérature, cette démarche est parfois surnommée exploration *et-si* (*what-if,* en anglais) [69] du modèle physique. Les applications de visualisation scientifique interactive sont, elles, un type particulier d'applications scientifiques où les codes de simulation sont itératifs, où les composants de visualisation peuvent retourner des données d'interaction au reste de l'application et où l'intégrité des données transmises entre deux composants peut être infléchie contre une communication plus rapide. Une architecture flot de données ou hybride se prête bien aux applications de réalité virtuelle si elle sait gérer des flux de données continus (*data streams*, en anglais) [3]. Un modèle de composants qui l'applique à la simulation scientifique doit permettre plusieurs degrés de synchronisation entre composants, accepter les cycles et être suffisamment simple -ou simplifiable- pour être à la portée de non informaticiens. Au prochain chapitre, nous présentons le modèle de flux de travaux que nous avons défini dans ce but.

Deuxième partie

Contribution

Chapitre 4

Modèle de composants

Sommaire

4.1 Genèse

La visualisation scientifique interactive telle que nous l'avons définie au chapitre 1 a beaucoup de points communs avec la réalité virtuelle tant au niveau des programmes qui la constituent (visualisation, interaction, physique, etc.) que de la performance qui en est attendue. Plutôt que d'être défini à partir de zéro ou à partir d'un système de gestion de flux de travaux, nous avons choisi de nous inspirer d'un modèle destiné à la réalité virtuelle. En l'occurrence, il s'agit de celui de FlowVR [1], un intergiciel pour concevoir et exécuter des applications distribuées haute-performance dont la convenance à la réalité virtuelle a été démontrée [2]. Il adopte une vision flot de données de l'application mais son modèle de composants n'a jamais été complètement formalisé.

FlowVR a été développé et est maintenu par l'équipe MOAIS de l'INRIA Rhône-Alpes et par l'équipe PRV du LIFO. Il s'est donc imposé comme moteur logiciel principal pour le projet de laboratoire virtuel FVnano unissant ces deux équipes. Le présent travail de thèse rentrant également dans le cadre de ce projet ANR, il était donc nécessaire que le modèle de composants qui en résulte soit compatible avec FlowVR. Notre modèle lui emprunte, par exemple, sa définition de composant itératif. Rien dans sa spécification ne l'empêche cependant d'être implémenté dans un autre système de flux de travaux.

4.2 Éléments de base

Nous présentons, dans cette section, les éléments de base constituant notre modèle de composants. La structure et le comportement de ces éléments ont été pensés pour remplir

des objectifs de performance et de cohérence scientifique inhérentes aux applications de visualisation scientifique interactive.

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. Une application est représentée, dans notre modèle, par un assemblage de ces éléments appelé *graphe d'application*.

4.2.1 Le composants

Un composant encapsule une tâche dans la/les chaîne(s) de traitement de l'application et est itératif par nature. Il peut être parallèle. 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, ce 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 4.1, 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-résultat à travers tous ses ports de sortie ainsi que, via la port e, un signal notifiant la fin d'une itération.

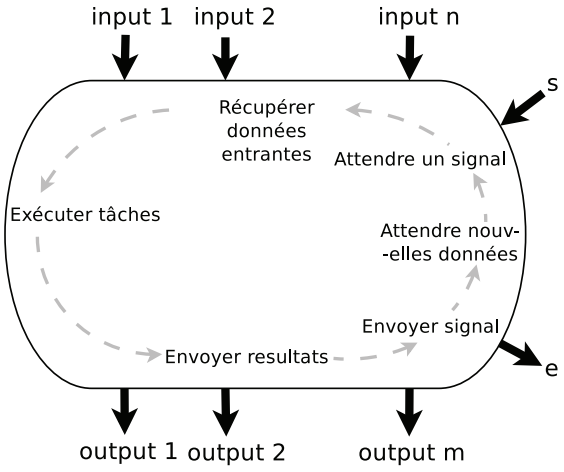


FIGURE 4.1 – Le cycle d'itération du composant.

Les ports de données d'un composant portent, chacun, 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, dans cette situation, est défini par l'utilisateur à l'implémentation.

Dans la suite de ce mémoire, 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.

4.2.2 Les connecteurs élémentaires

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 [4, 74] :

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 [74]. 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.

Ensuite, outre la prévention de la saturation, un modèle destiné aux applications de visualisation interactive doit aussi, parfois, permettre à un composant de fonctionner à sa propre fréquence. Pour ce faire, il doit lui éviter d'être contraint par la fréquence, plus faible, d'un autre composant qui l'alimente en messages. 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 en messages vides 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 4.2, sont les suivants :

- Le sFIFO est une connexion FIFO dans laquelle, pour éviter toute saturation, l'émetteur, avant d'itérer, attend sur son port s un signal de déclenchement généralement émis par le receveur. Ce connecteur impose une synchonicité 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 afin de désamorcer des interblocages circulaires. Ce dernier usage est montré à la section 5.3 de ce mémoire. Le comportement non-bloquant est indiqué par le préfixe "nb" ;
- 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.

On peut remarquer que, dans la figure 4.2, les connecteurs sont organisés selon deux critères. Cette approche rappelle celle des auteurs de Kepler [60] qui ont formulé leurs réalisateurs par des combinaisons de critères (voir le tableau 3.1 au chapitre 5) afin d'en faciliter le choix. Au prochain chapitre, nous expliquons comment l'utilisateur peut composer une application en se servant de ces deux paramètres.

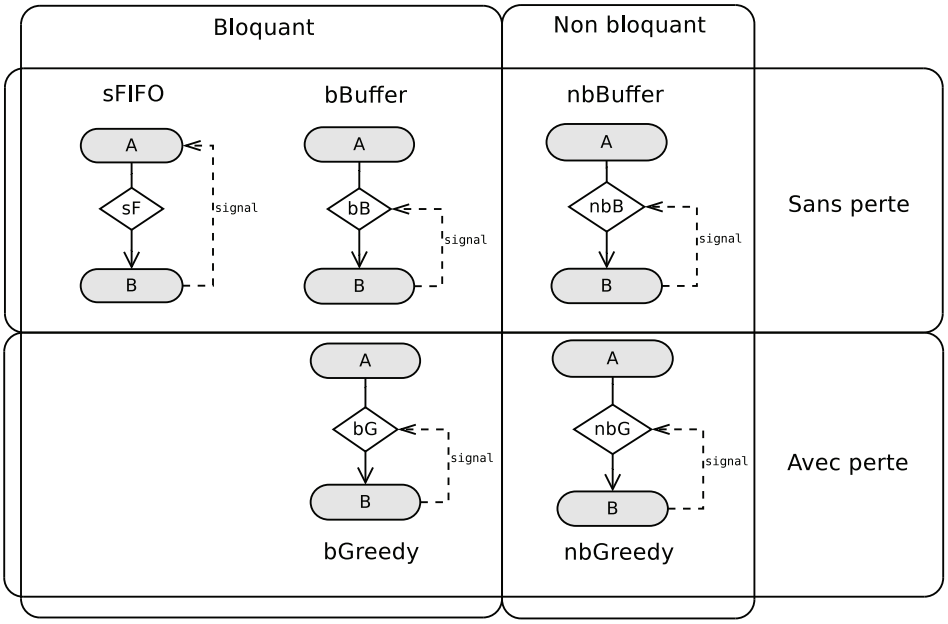


FIGURE 4.2 – Les cinq connecteurs.

4.2.3 Le régulateur

Le régulateur est un connecteur multi-canaux avec perte permettant de coordonner un ensemble de flots de messages. Sa politique de filtrage s'exprime par un ensemble de règles définies par l'utilisateur sur ses différents canaux. Ces règles sont des formules linéaires

sur les estampilles d'itération des messages. Formellement, un régulateur, illustré dans la figure 4.3, est un quintuplet $r = (n, \{I \cup \{s\}\}, O, F, b)$ avec :

- n son identifiant unique ;
- I son ensemble de ports d'entrée de données ;
- s un port de déclenchement ;
- O son ensemble de ports de sortie de données, de même taille que I avec lequel il est en bijection ;
- F l'ensemble de ses règles de filtrage ;
- b un booléen qui prend la valeur "vrai" si le régulateur doit être bloquant sur tous ses ports de sortie et "faux" s'il doit être non-bloquant sur tous ses ports de sortie.

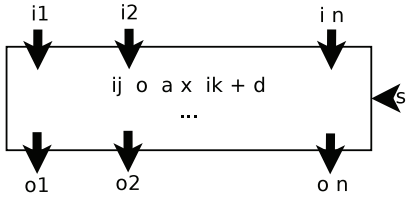


FIGURE 4.3 – Schéma du régulateur.

Nous emploierons, pour le régulateur, les mêmes notations que pour le composant lorsqu'elles s'appliquent. Nous noterons également $F(r)$ l'ensemble des règles de filtrage d'un régulateur r. Une règle de filtrage a la forme

$$i_k \circ \alpha \times i_l + \delta$$

avec $i_k, i_l \in I, \circ \in \{\leq, =, \approx\}$ et $\alpha, \delta \in \mathbf{N}$. L'opérateur $\approx$, utilisé avec $\delta > 0$, définit un écart absolu toléré de δ entre les deux opérandes i_k et i_l. Soit $M = \{m_1, \ldots, m_n\}$ un ensemble de messages en bijection avec $I(r)$ de telle sorte que M contient un et un seul message m_k provenant du port i_k de r à un instant t. On dit que M valide $f : i_k \circ \alpha \times i_l + \delta$ de $F(r)$ si $it(m_k) \circ \alpha \times it(m_l) + \delta$. M valide $F(r)$ s'il valide toutes les règles de $F(r)$.

En pratique, le régulateur empile, dans le buffer attaché à chacun de ses ports d'entrée, les messages arrivant à ce dernier. À chaque fois que le régulateur est déclenché, si une sélection de messages contenant un message de chaque buffer valide $F(r)$, chaque message de cette sélection est transmis par le port de sortie rattaché à son buffer. Lorsque plusieurs sélections sont possibles, c'est celle contenant les messages les plus récents qui est sélectionnée. Les messages antérieurs à ceux transmis sont, ensuite, supprimés des buffers.

4.2.4 Les liens

Les liens relient composants, connecteurs et régulateurs entre eux via leurs ports. Ils sont notés $(x.o, y.i)$ avec x et y des composants, connecteurs ou régulateurs 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 ou un régulateur est toujours requis pour définir la politique de communication entre deux composants. Par défaut, un port d'entrée ou de sortie d'un connecteur ou d'un régulateur 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, connecteurs et régulateurs n'attendent pas de signal de déclenchement pour effectuer leur toute première itération.

4.3 Graphe d'application

En présence des précédents éléments, une application est représentée par un graphe d'application. Ses sommets sont les composants, les connecteurs et les régulateurs. Ses arêtes sont les liens. Formellement, $\mathcal{G} = (\mathcal{C} \cup \mathcal{L} \cup \mathcal{R}, \mathcal{D} \cup \mathcal{T})$ définit un graphe d'application avec $\mathcal{C}$ un ensemble de composants, $\mathcal{L}$ un ensemble de connecteurs, $\mathcal{R}$ un ensemble de régulateurs, $\mathcal{D}$ un ensemble de liens de données et $\mathcal{T}$ un ensemble de liens de déclenchement.

La figure 4.4 montre un graphe d'application possible pour une application de visualisation scientifique interactive dans lequel nous utilisons différents types de connecteurs élementaires. Il est constitué de :

- une simulation découpée en deux composants de calcul ;
- un composant de visualisation qui ne propose qu'une interaction de type navigation ;
- un composant d'interaction relié à un périphérique et qui émet, à très haute fréquence, la position du curseur et l'état du bouton (pressé ou relâché) ;
- un composant générant, à partir la position du curseur et l'état du bouton, des forces à injecter dans 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 donc des bGreedy en sortie de ce composant afin de ne pas saturer les composants qu'il alimente. Ensuite, un nbBuffer relie le générateur de forces au premier composant de calcul. Puisque que le premier n'envoie pas de messages lorsque le bouton n'est pas pressé, nous préférons utiliser un connecteur non bloquant afin de ne pas bloquer le second. Nous choisissons, par ailleurs, un buffer car nous supposons qu'une discontinuité dans le flot de vecteurs de forces provoque des erreurs dans la simulation. Les deux étapes de la simulation, elles, sont reliées par des connecteurs sFIFO étant donné qu'elles relèvent de la même simulation. Enfin, un nbGreedy est utilisé entre la simulation et la visualisation car, la fréquence de la simulation pouvant varier, nous souhaitons avoir la visualisation la plus stable possible.

Pour la suite de ce mémoire, nous considérerons également les définitions suivantes :

Définition 5 (Chemin de données) *Un chemin de données dans le graphe d'application $\mathcal{G}$ est un chemin acyclique dans le graphe $(\mathcal{C} \cup \mathcal{L} \cup \mathcal{R}, \mathcal{D}) \subset \mathcal{G}$.*

Le chemin $P = (Interaction, c_2, Forces, c_3, Simulation_1, c_4, Simulation_2, c_6, Visualisation)$ est un exemple de chemin de données sur la figure 4.4. Nous définissons également la source et la destination d'un chemin de données.

Définition 6 (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 destination $dest(P)$ son sommet d'arrivée.*

Les composants *Interaction* et *Visualisation* sont, respectivement, la source et la destination du chemin P sur la figure 4.4. Nous définissons, enfin, le message résultat d'un chemin et la position d'un sommet dans un chemin.

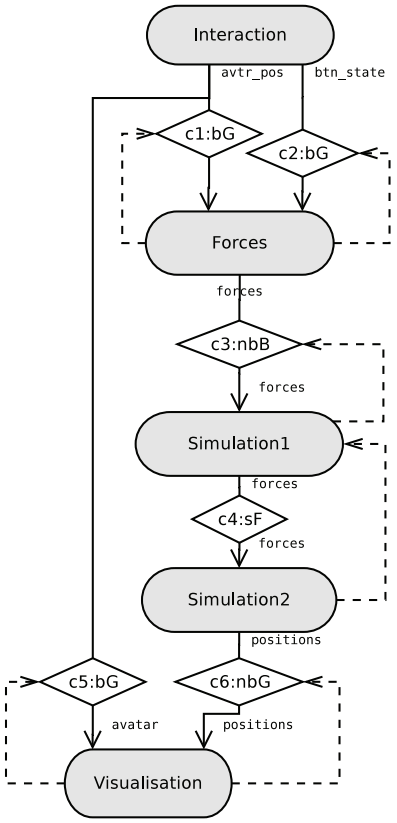


FIGURE 4.4 – Un graphe d'application.

Définition 7 (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 $ori_P(m)$.*

Définition 8 (Position sur un chemin de données) *$rank_p(x)$ désigne la position d'un sommet $x \in \{\mathcal{C} \cup \mathcal{L} \cup \mathcal{R}\}$ dans le chemin P.*

D'après cette définition, $rank_P(src(P)) = 1$ et $rank_P(dest(P)) = length(P)$ avec $length(P)$ le nombre de sommets dans P.

4.4 Conclusion

Nous avons, dans ce chapitre, présenté et formalisé un modèle flot de données pour applications de visualisation scientifique interactive. Il vise à faciliter l'intégration de codes patrimoniaux. Ceux-ci, pour les applications de visualisation interactive, 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 et ce, qu'elle ait changé ou pas ;
- Le module d'interaction est une boucle qui vérifie, à fréquence régulière, l'arrivée de nouvelles actions utilisateur.

C'est pour cela que l'itérativité est inhérente au composant du modèle. Ainsi, les codes patrimoniaux peuvent plus facilement se fondre dedans. Certains systèmes comme CUMULVS [50] ont, par le passé, adopté ce choix de conception.

L'application de visualisation scientifique interactive doit aussi être particulièrement performante. Elle peut, ainsi, non seulement traiter et afficher de gros volumes de données mais également répondre immédiatement aux actions de l'utilisateur et ne jamais rompre son immersion. La performance de l'application doit être favorisée par le modèle. Le choix d'une architecture par composants pour ce genre d'applications est dû à la nécessité d'intégrer des codes hétérogènes mais aussi à l'éventualité de les lancer de manière distribuée. Les composants les plus gourmands en ressources tels que la simulation et l'affichage peuvent ainsi bénéficier d'architectures matérielles spécifiques dédiées, respectivement, au calcul et au rendu. De plus, un modèle de composition flot de données est primordial. Ce choix est très lié au caractère itératif des composants qui fait que le composant s'auto-déclenche et peut donc fonctionner à son propre rythme. Une vision flot de données favorise l'indépendance des composants leur permettant, potentiellement, de fonctionner à leur plus hautes fréquences. Les schémas de communication que propose le modèle permettent de préserver cette désynchronisation mais aussi d'instaurer une certaine synchronisation lorsque les deux composants sont étroitement liés : par exemple, lorsque qu'un composant B a pour unique rôle de post-traiter les résultats d'un composant A avant de les transmettre au reste de l'application. Enfin, nonobstant la logique flot de données du modèle, les instruments de flot de contrôle que sont les liens de déclenchement peuvent être détournés de leur usage d'origine au bénéfice de motifs de synchronisation plus avancés.

La performance optimale souhaitée des applications ne doit pas occulter la fiabilité attendue des résultats d'une application scientifique. L'un des moyens d'obtenir de la performance dans un réseau flot de données est d'autoriser la perte de données mais une perte non contrôlée de données peut rapidement altérer la cohérence des résultats affichés. La notion de cohérence que nous entendons ici se définit entre plusieurs flots de données issus d'un ou de plusieurs composants et atteignant un même composant. Elle est quantifiée par le degré de synchronisation entre ces flots depuis leurs sources jusqu'à leur destination commune. Ce type de préoccupations en rapport avec la provenance des données est connu des concepteurs de flux de travaux scientifiques [91]. Il est alors nécessaire que notre modèle prévoie les 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.

Enfin, la composition doit, de manière générale, être facile. Il serait préférable pour l'utilisateur de décrire une application en renseignant des attributs locaux ou globaux plutôt qu'en ayant à apprendre la liste des connecteurs et leurs rôles. Dans cette optique, nous présentons, au prochain chapitre, la méthode de spécification lui permettant de construire une application par l'attribution de propriétés aux connexions, sans avoir à maîtriser les éléments présentés à la section 4.2. Les étapes de cette construction, dont seule la première n'est pas automatique, sont les suivantes :

1. Spécification de l'application par l'utilisateur ;
2. Génération du graphe d'application ;
3. Éventuellement modification du graphe d'application pour satisfaire des contraintes de cohérence ;
4. Débloquer les interblocages dans l'application ;
5. Éventuellement déplier le graphe d'application au cas où certains composants seraient parallèles.

Chapitre 5

Composition d'une application

Sommaire

5.1 Composition simple

La composition d'une application selon notre modèle de composants peut se faire par instantiation et liaison des différents éléments définis à la section 4.2. Nous proposons, toutefois, 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 le biais de ce graphe 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 marqués par le nombre d'instances souhaitées s'il est supérieur à 1. À défaut, ce nombre est 1 et signifie que le composant n'est pas parallèle. 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 messages 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 5.1. 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 4.2.1, un connecteur non bloquant et avec perte est indispensable en amont d'un composant interactif.

	Bloquant		**Non bloquant**	
Récepteur			**Interactif**	**Non interactif**
Perte	bGreedy,bBuffer,nbGreedy,nbBuffer		nbGreedy	nbGreedy,nbBuffer
Émetteur	**Interactif**	**Non interactif**		
Sans perte	bBuffer,nbBuffer	sFIFO, bBuffer,nbBuffer		nbBuffer

TABLE 5.1 – Règles de sélection automatique des connecteurs

Les figures 5.1 et 5.2 illustrent un schéma simplifié d'application de visualisation scientifique à travers son graphe de spécification (figure 5.1) puis le graphe d'application qui en est déduit (figure 5.2).

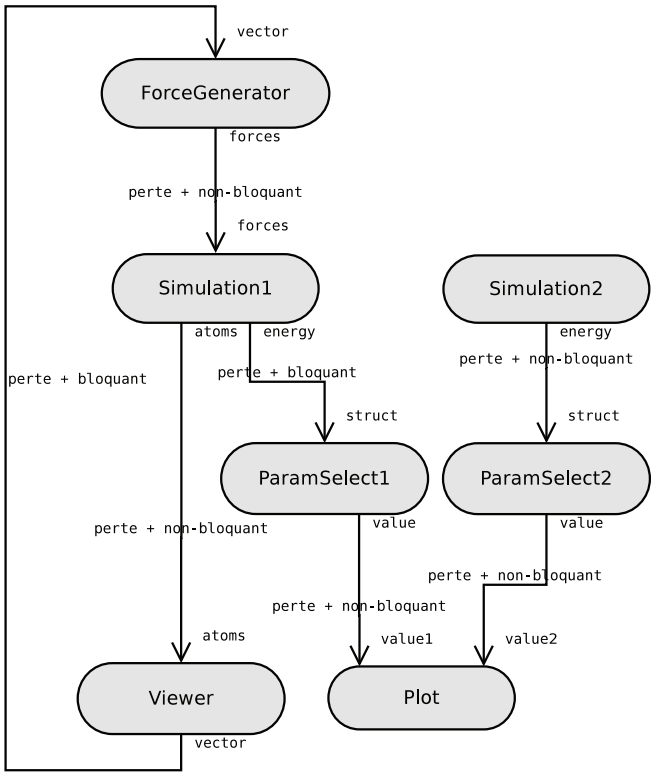


FIGURE 5.1 – Un graphe de spécification.

Si le graphe d'application contient des cycles d'interblocage, ils peuvent être résolus automatiquement selon la méthode décrite à la section 5.3. De plus, si l'application contient

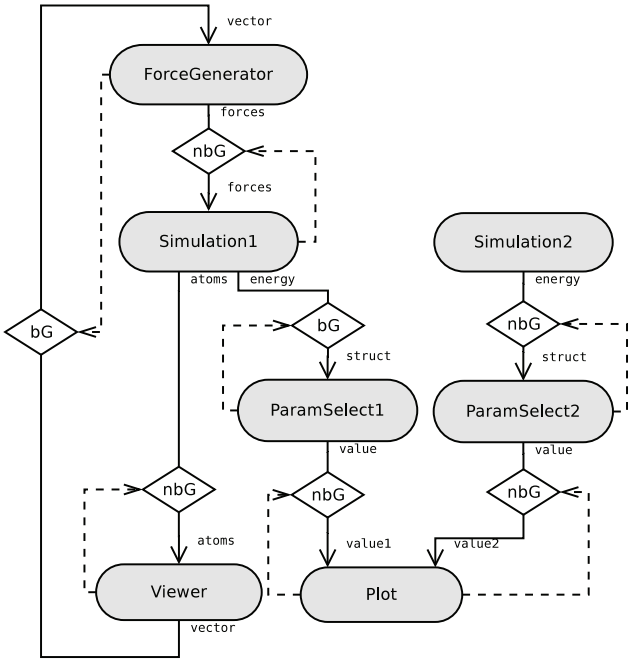


FIGURE 5.2 – Un graphe d'application.

des composants parallèles, le graphe d'application déplié pourra être déduit selon les règles décrites à la section 5.4. Avant cela, sont traitées les contraintes de cohérence exprimées à la spécification.

5.2 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. Ces contraintes 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 l'écart d'itération toléré entre les couples de messages m_1 et m_2 émis par deux ports de sortie et à l'origine de tous les messages arrivant simultanément à deux ports d'entrée d'un même composant. Formellement, la cohérence est définie ainsi :

Définition 9 (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*

$$A.o_{i \to C.i_1} \circ \alpha \times B.o_{j \to C.i_2} + \delta$$

avec $\circ \in \{\leq, =, \approx\}$ et $\{\alpha, \delta\} \in \mathbf{N}$ 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 $C.i_2$, $it(ori(m_1)) \circ \alpha \times it(ori(m_2)) + \delta$ 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 $\mathcal{G}$.

On parle de cohérence stricte lorsque l'égalité est demandée entre les itérations des messages issus des deux ports de sortie.

Définition 10 (Cohérence stricte) *Une contrainte de cohérence*

$$A.o_{i \to C.i_1} \circ \alpha \times B.o_{j \to C.i_2} + \delta$$

est dite stricte si $\circ =$ " $=$ ", $\alpha = 1$ *et* $\delta = 0$.

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 $C.i_2$.

FIGURE 5.3 – Visualisation scientifique interactive.

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. Lorsque l'application a pour objet d'afficher simultanément les résultats de deux simulations concurrentes afin de les comparer, la cohérence peut servir à placer ces résultats sur la même échelle de temps. La cohérence peut également être garante d'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 comme dans la situation représentée sur la figure 5.3, 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.

5.2.1 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 11 (Chemins frères) *ϕ_κ désigne l'ensemble des chemins impliqués dans une contrainte de cohérence $\kappa = A.o_{i \to C.i_1} \circ \alpha \times B.o_{j \to C.i_2} + \delta$. 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. L'approche utilisée est l'algorithme récursif de parcours de graphe en profondeur [17]. Pour chaque contrainte $\kappa = A.o_{i\to C.i_1} \circ \alpha \times B.o_{j\to C.i_2} + \delta$, l'algorithme est lancé à partir de $C.i_1$ et $C.i_2$ avec pour conditions d'arrêt la rencontre de, respectivement, $A.o_i$ ou $B.o_j$ ou bien un composant déjà visité différent de A et de B. Uniquement dans le premier cas, le chemin est stocké dans ϕ_κ. Une première vérification de la validité de la contrainte peut être opérée à cette étape. En effet, si aucun chemin n'est trouvé entre $A.o_i$ et $C.i_1$ (respectivement, entre $B.o_j$ et $C.i_2$), cela signifie qu'aucun message produit par $A.o_i$ (respectivement $B.o_j$) n'est à l'origine d'un flot de données arrivant à $C.i_1$ (respectivement, $C.i_2$). La cohérence avec l'autre port de sortie ne peut, alors, pas être établie.

Définition 12 (Validité d'une contrainte de cohérence) *Une contrainte de cohérence $\kappa = A.o_{i\to C.i_1} \circ \alpha \times B.o_{j\to C.i_2} + \delta$ n'est pas valide si $\nexists P \in \phi_\kappa$ tel que P relie $A.o_i$ à $C.i_1$ ou $B.o_j$ à $C.i_2$.*

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

Définition 13 (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 5.4 illustre le sous-graphe d'une cohérence κ imposée au graphe d'application de la figure 5.2.

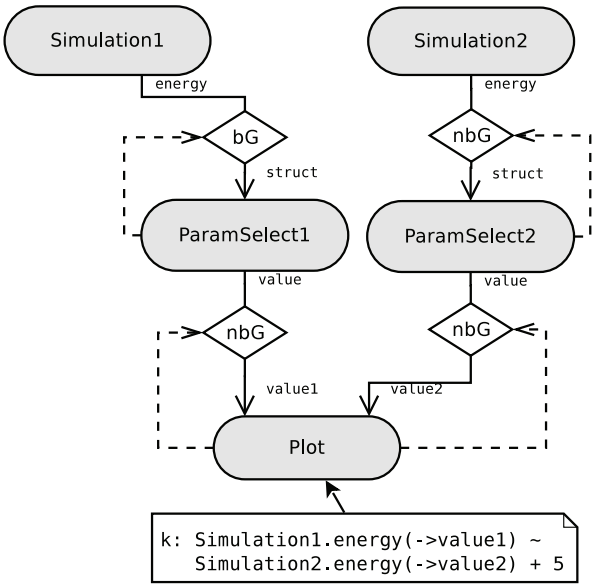


FIGURE 5.4 – Sous-graphe de cohérence.

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.

Définition 14 (Contraintes de cohérence jointes) *Deux contraintes de cohérence κ_1 et κ_2 sont dites jointes si $\exists c$ un connecteur tel que $c \in P_1, c \in P_2$ avec $P_1 \in \phi_{\kappa_1}$ et $P_2 \in \phi_{\kappa_2}$. Elles sont disjointes sinon.*

5.2.2 Transformations pour établir une cohérence stricte

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

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 15 (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.*

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.

Définition 16 (Jonction et connecteurs synchrones) *La jonction est un connecteur bGreedy, nbGreedy ou nbBuffer entre deux segments synchrones consécutifs. Les autres connecteurs -sFIFO et bBuffer- sont dits connecteurs synchrones.*

Des chemins frères entièrement synchrones -ne contenant pas de jonctions- sont naturellement cohérents mais leur performance est limitée. En effet, une conséquence de la propriété 1 est que les composants les plus lents dans un segment synchrone ralentissent ceux qu'ils alimentent. Les jonctions, instruments de désynchronisation par définition, rompent l'intégrité des flots de données qui les traversent et, donc, toute cohérence avec d'autres flots de données. Cependant, la conservation de la cohérence de chemins de données malgré la présence de jonctions reste possible et s'établit sur le sous graphe de cohérence. Le motif de composition général pour établir et maintenir une cohérence stricte entre deux chemins de données est représenté sur la figure 5.5. Pour obtenir une cohérence stricte κ, la procédure consiste à segmenter chaque chemin de ϕ_κ en une succession de segments synchrones et de jonctions de sorte que leurs nombres soient égaux entre tous les chemins de ϕ_κ. Comme le montre la figure 5.5, le motif établissant la cohérence stricte consiste, ensuite, à synchroniser deux par deux les jonctions des deux chemins afin que le passage de messages à travers ces zones critiques se fasse simultanément.

L'application de ce motif entre tous les couples de chemins frères contenus dans ϕ_κ assure la cohérence de l'ensemble du sous-graphe. Néanmoins, il est d'abord nécessaire que tous les chemins de ϕ_κ aient le même nombre de segments synchrones et le même nombre de jonctions. Une méthode de segmentation automatique va permettre d'aboutir à ce résultat. La flexibilité de la spécification observée dans le tableau 5.1 fait que différents connecteurs peuvent matérialiser une connexion. Plus précisément, les contraintes de blocage, de non blocage et de perte peuvent être abandonnées au bénéfice de la cohérence. En revanche, la contrainte de non perte, elle, n'est jamais relâchée. Concrètement, il est possible de transformer des sFIFO/bBuffer en nbBuffer et des bGreedy/nbGreedy en sFIFO, bBuffer ou nbBuffer tout en respectant la spécification.

Le 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 entre tous les chemins du sous-graphe et pas seulement entre ceux de ϕ_κ. Cela a pour effet de l'opérer de part et d'autre de chaque segment commun.

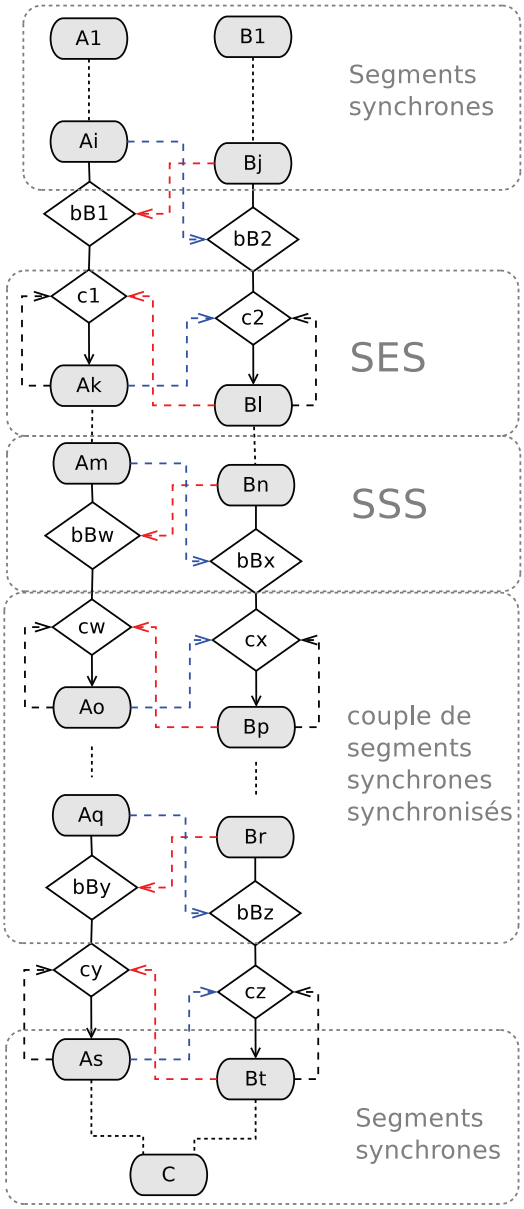


FIGURE 5.5 – Motif général pour la cohérence stricte.

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 5.1, 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 $\mathcal{G}_\kappa$. g_c 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 $Eq_{\mathcal{G}_\kappa} \cup Fix_{\mathcal{G}_\kappa} \cup Max_{\mathcal{G}_\kappa}$.

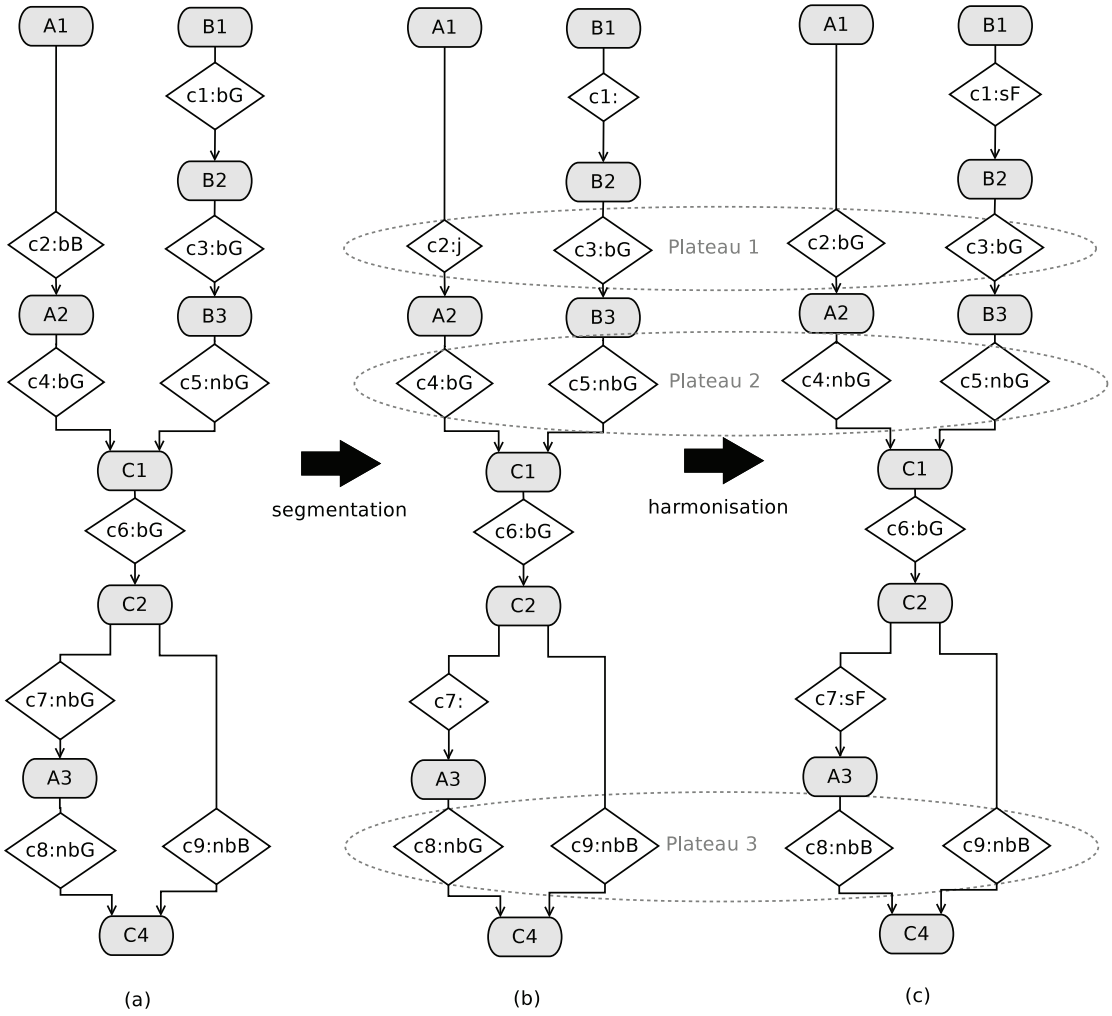


FIGURE 5.6 – Exemple de segmentation.

La figure 5.6b montre le résultat de la segmentation sur un exemple (figure 5.6a) 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).

Harmonisation des plateaux

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

Définition 17 (Plateau) *Soient $\mathcal{G}$ un graphe d'application, $\kappa = A.o_{i \to C.i_1} \circ \alpha \times B.o_{j \to C.i_2} + \delta$ 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, dans un sous-graphe commun à plusieurs cohérences, si des plateaux ont au moins une jonction commune, ils sont fusionnés en un seul.

Sur la figure 5.5, les couples de connecteurs $\{c_1, c_2\}$, $\{c_w, c_x\}$ et $\{c_y, c_z\}$ 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 une cohérence stricte. Comme expliqué plus loin à la section 5.2.3, il peut aussi servir à établir (en devenant un régulateur) une cohérence non stricte ou à maintenir une telle 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 17. 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. Sinon, nbGreedy s'il y a au moins un nbGreedy parmi les jonctions car, performance oblige, nous privilégions les nbGreedy aux bGreedy.

Des connecteurs synchrones remplacent les jonctions supprimées. Pour leur attribuer leurs types, 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.

La figure 5.6c montre le résultat de cette harmonisation des plateaux par rapport à la figure 5.6b. Sur l'exemple plus simple de la figure 5.4, la segmentation conservera les

quatre jonctions présentes et l'harmonisation des plateaux changera le connecteur bGreedy en nbGreedy.

Établissement de la cohérence stricte

Une fois les plateaux constitués et harmonisés, les motifs de synchronisation visibles sur la figure 5.5 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 18 (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 s'appuie sur l'assomption selon laquelle le temps de transfert d'un message ou d'un signal dans un lien est nul. 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 leurs segments respectifs et qu'elles soient donc de même type.

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, elle, un motif de composition permettant d'assurer la cohérence en émission de données de deux segments synchrones.

Définition 19 (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 démontrons, à présent, qu'un motif de composition constitué d'une succession de SSS et de SES tel qu'illustré sur la figure 5.5 est à même de satisfaire une contrainte de cohérence stricte $\kappa = (A_1.o_{i_{\rightarrow C.i_1}} = B_1.o_{j_{\rightarrow C.i_2}})$ 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 $B_1.o_j$ à $C.i_2$.

D'abord, les segments synchrones $S'_1 = (A_1 \dots A_i)$ et $S'_2 = (B_1 \dots B_j)$ 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 prouver 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 20 (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* sychronisés. *$[J * (S_1, S_2) * bB]^q$ exprime la composition de q segments synchronisés $J^1 * (S_1^1, S_2^1) * bB^1 * \dots * J^q * (S_1^q, S_2^q) * bB^q$.*

Nous commençons par démontrer qu'un couple de segments 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$^{\text{ème}}$ message. Un ensemble de séries de messages $\{M_1, \dots, M_n\}$ est dit synchronisé si $|M_1| = \dots = |M_n|$ et, $\forall i \in [1, |M_1|]$, $it(m_1^i) = \dots = it(m_n^i)$.

Théorème 1 *Soit $\mathcal{G}$ un graphe d'application et $(S_1, S_2) = J * (s_1, s_2) * bB$ un couple de segments 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 5.6, deux chemins frères peuvent s'intersecter avant leur destination commune. Ainsi, les couples de segments synchrones (S_1, S_2) -mais aussi (S''_1, S''_2)- peuvent être partiellement ou entièrement confondus. La succession $(S_1, S_2) = [(s_1, s_2)|s]^m$ de portions communes S et distinctes (s_1, s_2) préserve néanmoins la cohérence des flots des deux chemins et ce, en préservant le nombre et l'ordre de leurs messages. En effet, d'après la propriété 1, puisque S_1 et S_2 sont des segments synchrones, alors $it(m_1^m) = it(ori_{S_1}(m_1^m))$ et $it(m_2^m) = it(ori_{S_2}(m_2^m))$. Or, si les premiers composants de S_1

et de S_2 sont synchronisés -par les séries de messages synchronisées qu'ils reçoivent- ou s'ils sont confondus ($src(S_1) = src(S_2)$), alors $it(ori_{S_1}(m_1^m)) = it(ori_{S_2}(m_2^m))$.

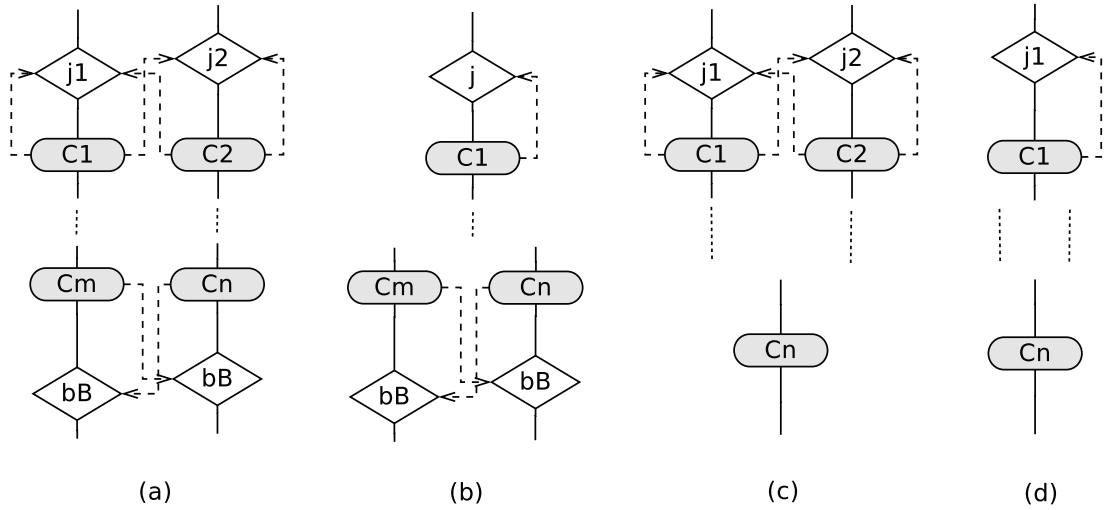


FIGURE 5.7 – Les quatre motifs de synchronisation entre deux segments sécants.

Dans le cas général, un couple de segments synchrones (S_1, S_2) est précédé d'un couple de jonctions et suivi d'un couple de bBuffers. 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 5.7a 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 commune. La spécification de la SES prend cette éventualité en compte. Le motif $j * (S'_1, S'_2) * bB$ illustré par la figure 5.7b assure cette synchronicité lorsque les deux segments commencent par le même composant. La SES est, ici, implicite car le même message alimente les deux segments. Le motif $J * (S'_1, S'_2)$ de la figure 5.7c 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.

Enfin, sur la figure 5.7d, les deux synchronisations sont implicites car les deux segments commencent par le même composant et se terminent par le même composant. On peut noter ce motif $j * (S'_1, S'_2)$. Comme les motifs des trois autres figures, il répond au théorème 1.

La synchronicité d'une suite de couples de segments synchronisés peut ensuite être déduite.

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 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 stricte 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, chacun, par un segment synchrone (*S''_1 *et* S''_2*) maintenant, d'après la propriété 1, la cohérence déjà établie entre eux.*

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 10.

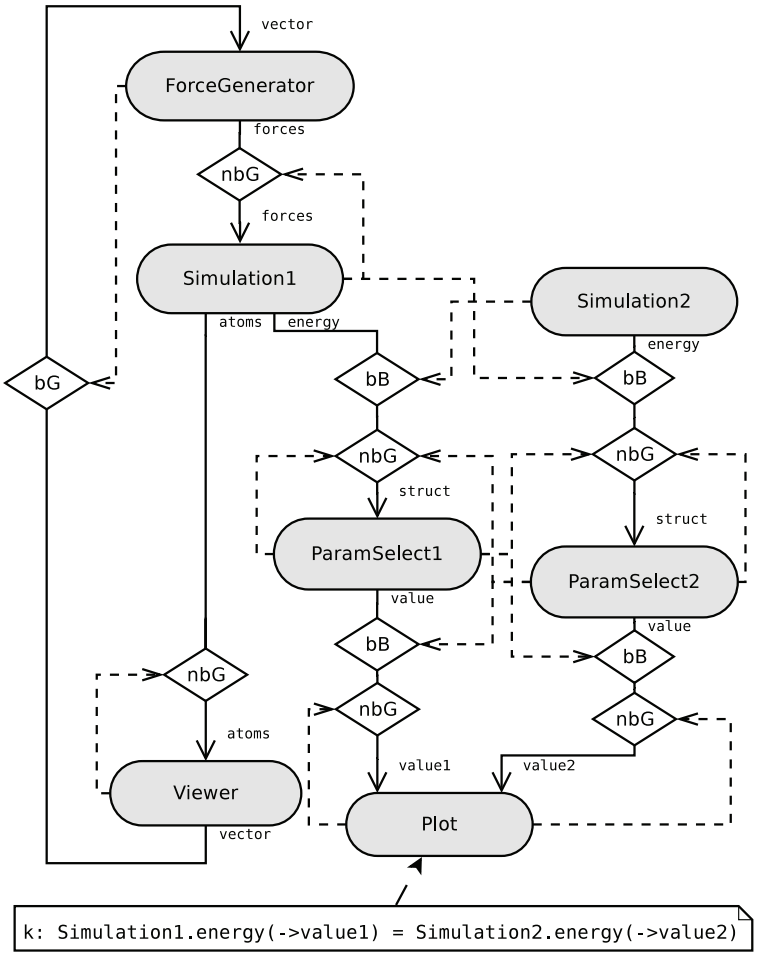


FIGURE 5.8 – Application simple avec cohérence stricte établie.

La figure 5.8 montre le graphe d'application final spécifié dans la figure 5.1 après mise en place de ce motif pour réaliser une cohérence stricte.

5.2.3 Transformations pour établir une cohérence non stricte

L'établissement d'une contrainte de cohérence non stricte repose sur un schéma semblable à celui de la cohérence stricte. Il s'agit, comme le montre la figure 5.11, également

d'une succession de SSS et de SES. La première SSS de ce motif est, toutefois, faite d'un régulateur et non de deux jonctions. Le rôle de ce régulateur est d'établir la cohérence entre les deux flots de données. Le reste du motif conserve cette cohérence en assurant la synchronisation des deux flots.

La mise en place de ce motif suit les mêmes étapes que pour la cohérence stricte : segmentation, harmonisation et synchronisation. Toutefois, tandis que l'intersection de chemins soumis à différentes contraintes strictes aboutit à une synchronisation stricte de l'ensemble, la conciliation de plusieurs contraintes non strictes entres elles et, éventuellement, avec des contraintes strictes nécessite plus d'attention. Par exemple, si deux sous graphes de deux contraintes non strictes ont des portions communes, le flot de messages traversant cette portion doit aller dans le sens des deux cohérences. Précisément, si un chemin P est impliqué dans deux contraintes de cohérence, il peut successivement traverser deux régulateurs, un pour chaque contrainte, et il ne faut pas, alors, que le second fausse la cohérence du premier en établissant sa propre cohérence. D'autres conflits potentiels doivent également être anticipés et résolus plus tôt, au moment de la segmentation. Nous illustrons, dans la figure 5.9, le motif de composition général en présence de deux contraintes de cohérence non strictes jointes. La combinaison des motifs de composition pour de telles contraintes dépend des positions relatives de leurs régulateurs respectifs.

Combinaison à régulateur commun Les régulateurs sont placés au niveau des premières connexions avec perte des chemins frères. Lorsque ces connexions appartiennent aux chemins de deux contraintes de cohérence différentes, les régulateurs de ces contraintes peuvent se confondre en un seul. Les messages émis par ce régulateur doivent satisfaire les deux contraintes à la fois. Le régulateur R_1 de la figure 5.9 est ainsi censé établir les cohérences κ_1 et κ_2 simultanément et donc, ne laisser passer que des messages qui satisfont les deux contraintes en même temps. Avec ce dispositif, si deux règles inscrites dans le régulateur et appartenant chacune à l'une des contraintes sont contradictoires, le régulateur se retrouve bloqué. L'utilisateur devra alors revoir sa spécification quitte redéfinir l'une des contraintes de cohérence.

Combinaison à régulateurs séparés Dans ce motif, les deux régulateurs sont distincts. Le second doit à la fois maintenir l'égalité entre les itérations des messages provenant du premier -afin de maintenir la cohérence qu'il a établi- et imposer les règles de filtrage de sa propre cohérence. La figure 5.9 illustre cette situation à travers le régulateur R_1 établissant la cohérence κ_1 et le régulateur R_2 dont le rôle est à la fois d'établir la cohérence κ_3 et de préserver la cohérence κ_1. Dans ce cas, on dit que le régulateur R_2 est le *régulateur principal* de κ_3 et qu'il est un *régulateur secondaire* pour κ_1. Là encore, une contradiction dans les règles contenues dans R_2 peut entraîner un blocage de ce dernier.

Segmentation des chemins et harmonisation des plateaux

La segmentation des chemins d'un sous-graphe soumis à une cohérence non stricte égalise le nombre des jonctions entre l'ensemble des chemins du sous-graphe tout en essayant de préserver au maximum celles initialement présentes.

Les exigences sur le sous-graphe sont un peu plus nombreuses que sur un sous graphe n'ayant que des contraintes strictes. En effet, nous savons qu'une cohérence non stricte nécessite l'emploi d'un régulateur. Or, celui-ci étant avec perte, il sera placé, sur chaque chemin, en lieu et place d'une connexion autorisant la perte. Dès lors, si l'un des chemins d'une cohérence non stricte κ ne contient pas de connecteur avec perte, il ne pourra pas

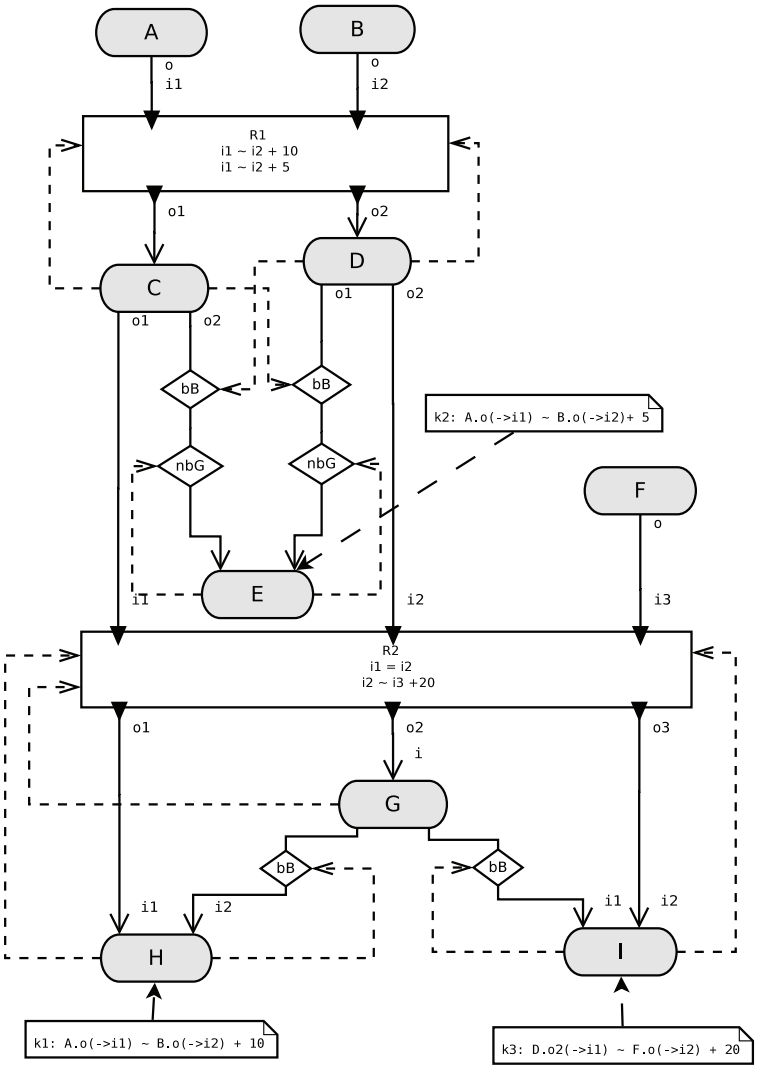


FIGURE 5.9 – Graphe d'application avec des contraintes de cohérence jointes.

être rendu cohérent avec les autres éléments de ϕ_κ. Dès lors, nous pouvons déduire que la contrainte est invalide puisqu'elle ne pourra pas s'appliquer à tous les chemins de ϕ_κ telle que l'exige la définition 9.

De surcroît, un régulateur établissant une cohérence entre deux chemins frères doit être placé, comme le montre la figure 5.9, avant toute intersection entre ces deux chemins. Autrement, leurs flots de données deviennent confondus et aucune comparaison ne peut être opérée. Il est également nécessaire que pour qu'un chemin soit traité, il contienne un connecteur avec perte avant tout segment commun avec l'un de ses chemins frères. À partir de ces conditions, nous pouvons formuler la validité d'une contrainte de cohérence :

Définition 21 (Validité d'une contrainte de cohérence non stricte) *La contrainte de cohérence non stricte κ est considérée valide ssi $\forall P, P' \in \phi_\kappa, \exists c \in P$ un connecteur avec perte tel que $\nexists C \in P \cap P'$ un composant tel que $rank_P(C) < rank_P(c)$.*

La première jonction avec perte d'un chemin impliqué dans une contrainte de cohérence est, ensuite, mise en évidence.

Définition 22 (γ d'un chemin de données) *Soient une contrainte de cohérence κ et $P \in \phi_\kappa$. Nous notons γ_P le connecteur avec perte dans P tel que $\nexists c \in P$ un connecteur avec perte tel que $rank(c) < rank(\gamma_P)$.*

Le système linéaire grâce auquel est résolue la segmentation est semblable à celui utilisé pour la segmentation d'un sous-graphe de cohérence stricte : $Eq_{\mathcal{G}_\kappa} \cup Fix_{\mathcal{G}_\kappa} \cup Max_{\mathcal{G}_\kappa}$. Toutefois, il est ici crucial d'anticiper le placement des régulateurs et, en particulier, le fait qu'il doivent être précédés d'un segment synchrone S sur chaque chemin. Autrement, nous aurions $it(m) \neq it(ori_S(m))$ pour un message m atteignant le régulateur et il ne serait plus possible d'exprimer de contraintes sur $it(ori_S(m))$ dans ce régulateur. En conséquence, il ne faut pas qu'il y ait d'autres jonctions au dessus de γ_P dans un chemin P impliqué dans une cohérence non stricte. Dès lors, pour tout chemin P, les variables de tous les connecteurs c de rang inférieur à γ_P sont fixées à $v_c = 0$ dans le système linéaire. Formellement, l'ensemble des équations supplémentaires devient donc $Fix_{\mathcal{G}_\kappa} = \bigcup_{c \in \mathcal{L'}_{\mathcal{G}_\kappa}, c' \in \mathcal{L''}_{\mathcal{G}_\kappa}} \{v_c = 1, v'_c = 0\}$ avec $\mathcal{L'}_{\mathcal{G}_\kappa} \subset \mathcal{L}_{\mathcal{G}_\kappa}$ l'ensemble des connecteurs c de $\mathcal{G}_\kappa$ ayant, comme récepteurs, un composant interactif et $\mathcal{L''}_{\mathcal{G}_\kappa}$ l'ensemble des connecteurs c' de $\mathcal{G}_\kappa$ tels que $rank(c') < rank(\gamma_P)$ avec $c' \in P, P \in \phi_\kappa$ et κ une contrainte de cohérence non stricte.

La fonction objectif reste identique à celle de la cohérence stricte : $Max_{\mathcal{G}} = Maximize\ (\Sigma_{c \in \mathcal{J}_{\mathcal{G}}}(2^{g_c} \times v_c))$ où $\mathcal{J}_{\mathcal{G}}$ est l'ensemble des jonctions initialement présentes dans $\mathcal{G}$ et g_c un booléen prenant la valeur 1 si c est avec perte. Contrairement aux contraintes non strictes, les connecteurs avec perte ne sont pas nécessaires à l'établissement d'une cohérence stricte. À cause de cela, la présence de contraintes strictes dans le même sous graphe que des contraintes non strictes peut négativement influencer la transformation des chemins de ces dernières. Le problème est illustré par l'exemple de la figure 5.10. Il représente un sous-graphe comprenant la combinaison d'une contrainte non stricte κ_1 avec une contrainte stricte κ_2 avant (figure 5.10a) et après (figure 5.10b) segmentation et harmonisation. Sur cet exemple, soient $\gamma_{P_1} = c_4$ et $\gamma_{P_2} = c_5$ avec $\phi_{\kappa_1} = \{P_1, P_2\}$. À cause de $Fix_{\mathcal{G}_{\kappa_1}}$, nous aurons, suite à la segmentation, $v_{c_1} = 0$ et $v_{c_2} = 0$. En conséquence, afin d'obtenir l'égalité du nombre de jonctions sur chaque chemin, c_3 serait transformé en connecteur synchrone si la segmentation ne privilégiait pas les Greedies. Cela aurait pour répercussion la formation d'un plateau $\{c_4, c_5, c_6\}$ dont le type serait, après harmonisation, nbBuffer forçant, ainsi, κ_1 à être stricte : $Max_{\mathcal{G}} = Maximize(v_{c_1} + v_{c_2} + v_{c_3} + v_{c_4} + v_{c_5} + v_{c_6} + v_{c_10}) = 4 = v_{c_4} + v_{c_5} + v_{c_6} + v_{c_10}$. La pondération des connecteurs avec perte dans la fonction objectif évite cette situation en favorisant la formation d'un plateau avec perte : $Max_{\mathcal{G}} = Maximize(v_{c_1} + v_{c_2} + 2 \times v_{c_3} + 2 \times v_{c_4} + 2 \times v_{c_5} + v_{c_6} + v_{c_10}) = 7 = 2 \times v_{c_3} + 2 \times v_{c_4} + 2 \times v_{c_5} + v_{c_10}$.

La figure 5.10b montre, ensuite, le sous-graphe formé par la segmentation puis harmonisation du sous-graphe de la figure 5.10a. La phase d'harmonisation se déroule selon la logique décrite à la section 5.2.2.

Placement du régulateur

Après harmonisation des plateaux, l'étape suivante consiste à remplacer les jonctions de chaque plateau régulateur par un régulateur. Un plateau régulateur est un plateau contenant les premières jonctions avec perte d'une contrainte de cohérence non stricte.

Définition 23 (Plateau régulateur) *Un plateau π est un plateau régulateur dans un graphe d'application $\mathcal{G}$ ssi $\exists \kappa \in \mathcal{K}_{\mathcal{G}}$ telle que $\forall P \in \phi_\kappa, \gamma_P \in \pi$. π est alors appelé plateau principal de κ et noté π_κ.*

Le rôle des régulateurs est d'appliquer les contraintes de cohérence sur les chemins qui les traversent. D'après la définition 9, ces contraintes sont formulées sur les sources des

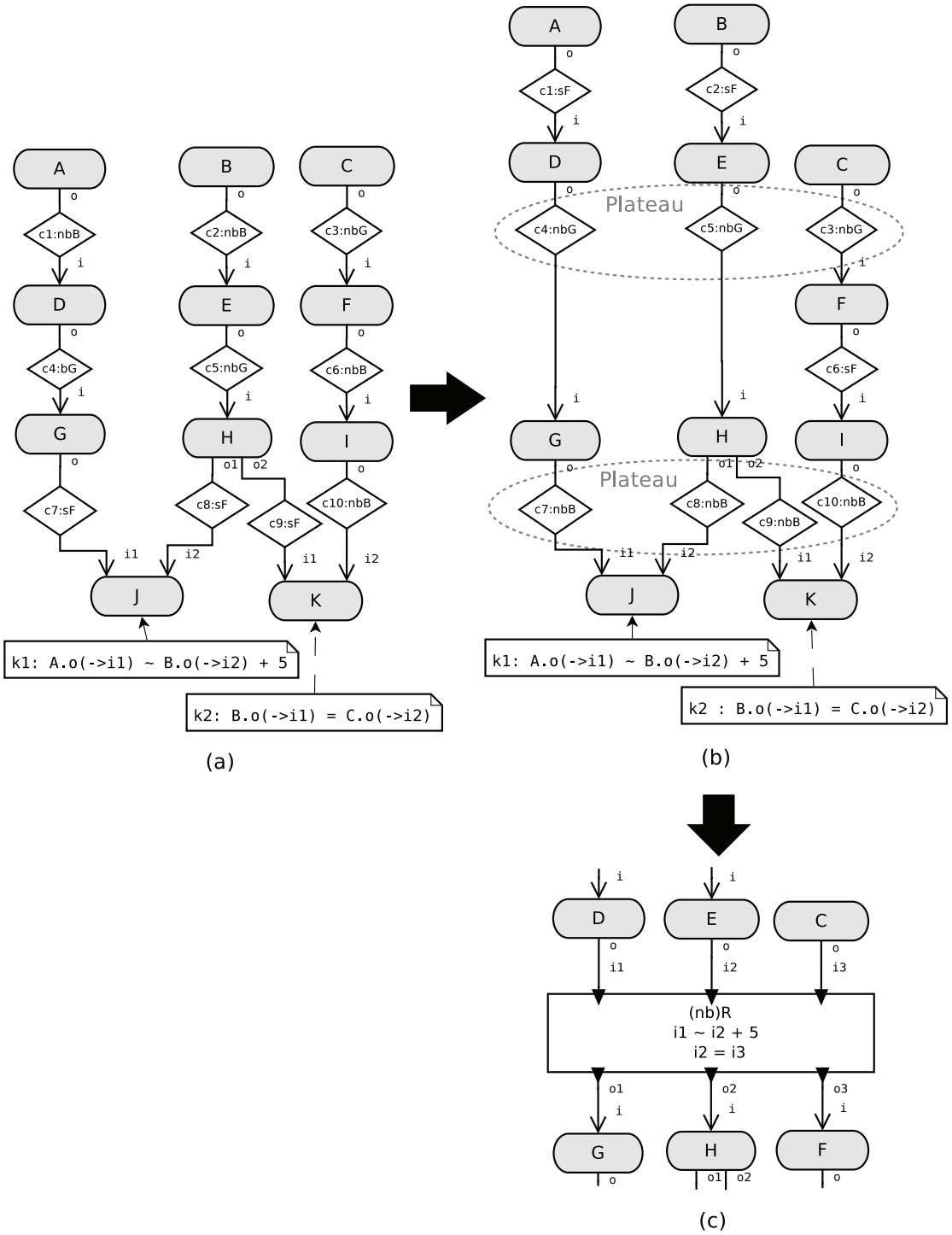


FIGURE 5.10 – Combinaison d'une contrainte non stricte et d'une contrainte stricte.

chemins impliqués dans la cohérence tandis que les règles contenues dans le régulateur s'appliquent à ses ports d'entrée. Une transformation élémentaire permet de déduire les règles de chaque régulateur à partir des contraintes de cohérence dans lesquelles il intervient.

Le remplacement d'un plateau par un régulateur se fait grâce à l'opération de substitution qui va consister à numéroter les connecteurs du plateau de 1 à n et à replacer chaque lien allant vers le port d'entrée du connecteur numéro i par un lien allant sur le port d'entrée numéro i du régulateur et symétriquement pour les ports de sortie.

Définition 24 (Substitution de plateau) *Soient G un graphe d'application contenant le plateau π composé des connecteurs $j_1, \ldots, j_n$, r un régulateur à n ports de données $i_1, \ldots, i_n$ et la bijection $\sigma_r : \{j_1, \ldots, j_n\} \to \{r.i_1, \ldots, r.i_n\}$ telle que $\sigma_r(j_k) = r.i_k \ \forall k \in [1, n]$. La substitution $\sigma_r(\pi)$ du plateau π dans le graphe G par le régulateur r fait que le régulateur r remplace les connecteurs $j_1, \ldots j_n$ de la manière suivante ($X.x$ et $Y.y$ sont respectivement des ports de sortie et d'entrée de composants X et Y quelconques) :*

- *chaque lien $(X.x, j_k.i)$ de G ($k \in [1, n]$) a été remplacé par le lien $(X.x, r.i_k)$ tel que $r.i_k = \sigma_r(j_k)$;*
- *chaque lien $(j_k.o, Y.y)$ de G ($k \in [1, n]$) a été remplacé par le lien $(r.o_k, Y.y)$ tel que $r.o_k$ est le port de sortie associé au port d'entrée $r.i_k$.*

La figure 5.10c illustre la substitution du premier plateau de la figure 5.10b.

Les règles du régulateur peuvent ensuite être définies en fonction de ses ports d'entrée :

Définition 25 (Règles d'un régulateur principal) *Soient π_κ le plateau principal d'une contrainte de cohérence κ dans le graphe d'application G et $\sigma_r(\pi_\kappa)$ la substitution de ce plateau par le régulateur r. Les règles placées dans ce régulateur pour κ est l'ensemble des règles :*

$$f : \sigma_r(\gamma_{P_1}) \circ_\kappa \alpha_\kappa \times \sigma_r(\gamma_{P_2}) + \delta_\kappa$$

telles que $\exists P_1, P_2 \in \phi_\kappa, P_2 \in \phi_\kappa(P_1)$ et $\gamma_{P_1}, \gamma_{P_2} \in \pi_\kappa$.

Comme expliqué précédemment, les régulateurs qui sont principaux pour certaines contraintes de cohérences peuvent également jouer un rôle secondaire de maintien pour d'autres. Là encore, la déduction des règles de filtrage assurant ce maintien est directe.

Définition 26 (Règles d'un régulateur secondaire) *Soient π un plateau qui n'est pas principal pour la contrainte de cohérence κ dans le graphe d'application G et $\sigma_r(\pi)$ la substitution de ce plateau par le régulateur r. Les règles placées dans ce régulateur pour κ est l'ensemble des règles :*

$$f : \sigma_r(j_1) = \sigma_r(j_2)$$

telles que $j_1, j_2 \in \pi$ sont deux jonctions.

Le régulateur R_2 de la figure 5.9 donne un exemple de ces deux types de règles.

Les règles de filtrage étant de simples équations linéaires, une vérification sémantique peut être opérée sur l'ensemble F de chaque régulateur r afin de détecter les contradictions pouvant causer le blocage du régulateur. Une combinaison de contraintes dans lesquelles intervient r peut être invalidée si cette analyse révèle une telle contradiction.

Préservation de la cohérence

Une fois le régulateur placé, l'objectif est de maintenir, jusqu'au composant destination, la cohérence établie. Cela est réalisé, comme le montre la figure 5.11, grâce au motif de

synchronisation assurant une cohérence forte. Nous étendons ici la définition 18 de la SES par une variante dans laquelle le plateau et non plus constitué de deux jonctions mais d'un régulateur r. Les liens de déclenchement croisés deviennent alors $(C_1^1.e, r.s)$ et $(C_2^1.e, r.s)$. Le fait que le régulateur doive être déclenché par les deux composants qu'il dessert ainsi que le fait qu'il émette toujours un message via chacun de ses ports à chaque fois garantissent la synchronicité des flots de messages atteignant C_1 et C_2 (A_k et B_l dans la figure 5.11).

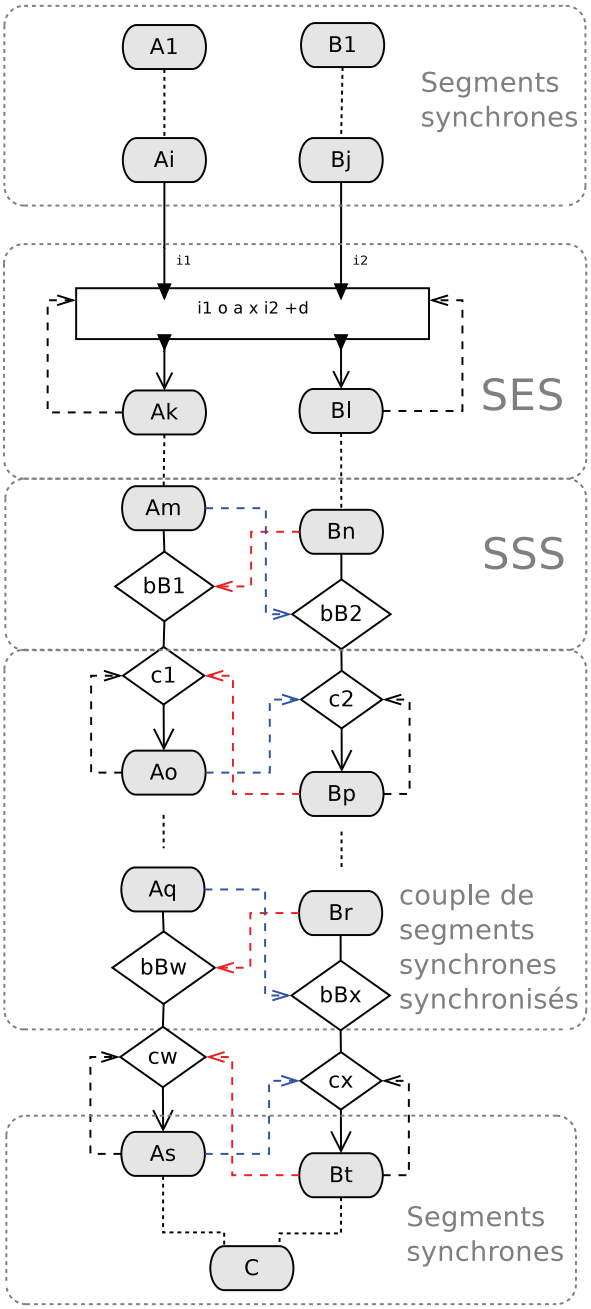


FIGURE 5.11 – Motif général pour la cohérence non stricte.

En maintenant la synchronisation des flots de données émis par le régulateur, nous maintenons également la cohérence qu'il a instauré.

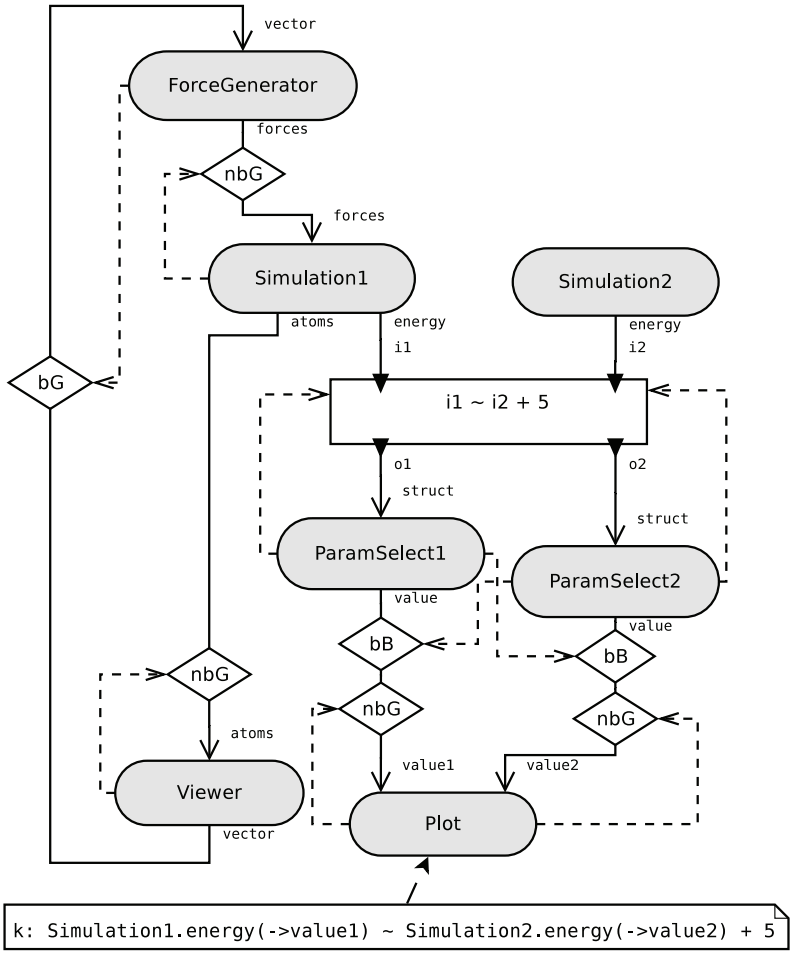


FIGURE 5.12 – Application simple avec cohérence non stricte établie.

Théorème 4 *Soient P_1 et P_2 deux chemins frères relativement à la contrainte de cohérence $\kappa = A.o_{i_{\rightarrow C.i_1}} \circ \alpha \times B.o_{j_{\rightarrow C.i_2}} + \delta$. Si m_1 et m_2 sont 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 $it(ori_{P_1}(m_1)) \circ \alpha \times it(ori_{P_2}(m_2)) + \delta$.*

Preuve 4 *Soient $(P_1, P_2) = (S_1^0, S_2^0) * r * (S_1', S_2') * bB * (P_1', P_2') * (S_1'', S_2'')$ deux chemins frères relativement à une contrainte de cohérence $\kappa = A.o_{1\rightarrow C.i_1} \circ \alpha \times B.o_{2\rightarrow C.i_2} + \delta$ tels que :*

- *r est un régulateur ;*
- *$(P_1', P_2') = [J * (S_1, S_2) * bB]^n$;*
- *S_1'', S_2'' deux segments synchrones aboutissant, respectivement, à $C.i_1$ et $C.i_2$.*

D'abord, les segments synchrones $S_1^0 = (A_1 \dots A_i)$ et $S_2^0 = (B_1 \dots B_j)$ garantissent, d'après la propriété 1, que pour tout message m_{A_i} émis par A_i, $it(m_{A_i}) = it(ori_{S_1^0}(m_{A_i}))$ et que pour tout message m_{B_j} émis par B_j, $it(m_{B_j}) = it(ori_{S_2^0}(m_{B_j}))$. Dès lors, la cohérence établie par le régulateur sur les messages reçus de A_i et B_j se reflète également sur les messages émis par $A_1.o$ et $B_1.o$ qui sont à leur origine. La sémantique du régulateur, décrite dans la section 4.2.3, assure, ensuite, que les messages m_{A_i} et m_{B_j} transmis par le régulateur satisfont la contrainte $it(m_{A_i}) \circ \alpha \times it(m_{B_j}) + \delta$ et donc, d'après le paragraphe précédent, la contrainte $it(ori_{S_1^0}(m_{A_i})) \circ \alpha \times it(ori_{S_2^0}(m_{B_j})) + \delta$. Le régulateur

est immédiatement suivi, dans chaque chemin, d'un segment synchrone. Une SSS maintient la cohérence des couples de messages m_{A_m}, m_{B_n} sortants de ces deux segments. À ce stade, $it(ori_{S_1^0}(m_{A_m})) \circ \alpha \times it(ori_{S_2^0}(m_{B_n})) + \delta$. Il s'agit, ensuite, de prouver 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.

Soient m_1 et m_2 deux messages lus à la même itération par C sur, respectivement, ses ports i_1 et i_2. D'après le théorème 3, $it(ori_{P'_1}(m_1)) = it(ori_{P'_2}(m_2))$. Or, à cause du motif de SSS $(S'_1, S'_2) * bB$, nous savons que le couple de messages $ori_{P'_1}(m_1), ori_{P'_2}(m_2)$ vérifie la contrainte de cohérence κ. Nous avons donc $ori_{P'_1}(m_1) \circ \alpha \times ori_{P'_2}(m_2) + \delta$. Enfin, S_1 et S_2 étant des segments synchrones, nous savons que pour tous messages m'_1 et m'_2 atteignant r, $it(m'1) = it(ori_{S_1}(m'_1)) = it(ori_{P_1}(m'_1))$ et $it(m'2) = it(ori_{S_2}(m'_2)) = it(ori_{P_2}(m'_2))$. Nous pouvons alors conclure que $ori_{P_1}(m_1) \circ \alpha \times ori_{P_2}(m_2) + \delta$.

Ce théorème prouve que la cohérence κ est établie et maintenue entre un couple de chemins frères par le motif de composition proposé.

La figure 5.12 illustre le graphe d'application final de l'application spécifiée sur la figure 5.1 et soumise à une contrainte de cohérence non stricte.

5.3 Déblocage des cycles synchrones

Le comportement du composant décrit dans la section 4.2.1 impose que le composant attende que tous ses ports d'entrée de données connectés soient alimentés avant d'itérer. Un interblocage peut, alors, se produire si un cycle synchrone est formé dans l'application.

Définition 27 (Cycle synchrone) *Dans un graphe d'application $\mathcal{G}$, un cycle synchrone est un chemin de données P tel que $src(P) = dest(P)$ et $\nexists c \in P$ un connecteur non bloquant.*

La figure 5.13 représente un graphe d'application comprenant des motifs de cohérence et plusieurs cycles synchrones non résolus. N'importe quel composant à l'intérieur d'un cycle synchrone se retrouve, pour itérer, indirectement dépendant d'un message qu'il doit lui-même produire. Pour débloquer un tel cycle, il suffit de transformer l'un de ses connecteurs en connecteur non bloquant. Notre approche est de minimiser les changements en commençant par transformer les connecteurs impliqués dans le plus grand nombre de cycles synchrones. Cependant, si des motifs de cohérence sont déjà en place dans l'application, il ne faut pas que ces modifications les faussent. En conséquence, plutôt que de compter les occurrences des connecteurs parmi tous les cycles élémentaires trouvés -cycles ne contenant pas d'autres cycles, nous comptons celles des plateaux, des régulateurs et des connecteurs indépendants. À l'issue de cette recherche, les plateaux bloquants ayant le plus de connecteurs impliqués dans des cycles synchrones seront changés en priorité. Sur la figure 5.13, le plateau formé par les jonctions $\{c_{15}, c_{16}\}$ est impliqué dans le cycle 4 via le connecteur c_{15} et dans le cycle 5 via le connecteur c_{16}. Nous privilégierons son changement en nbGreedy au détriment des trois connecteurs c_{12}, c_{14} et c_{17}. Par ailleurs, les bBuffer additionnels des SES, eux, ne sont jamais modifiés afin de préserver la cohérence du motif.

Les régulateurs, puisque traversés par plusieurs chemins, sont, également, susceptibles de débloquer un certain nombre de cycles en devenant non bloquants. C'est le cas du régulateur R de la figure 5.13 qui, en devenant non bloquant, permettrait de débloquer les cycles 2 et 6. Observons que le changement global d'un plateau ou d'un régulateur de l'état bloquant à celui de non bloquant préserve leurs politiques de synchronisation sur les chemins qu'ils contrôlent. En revanche, l'algorithme ne s'autorise pas à transformer les

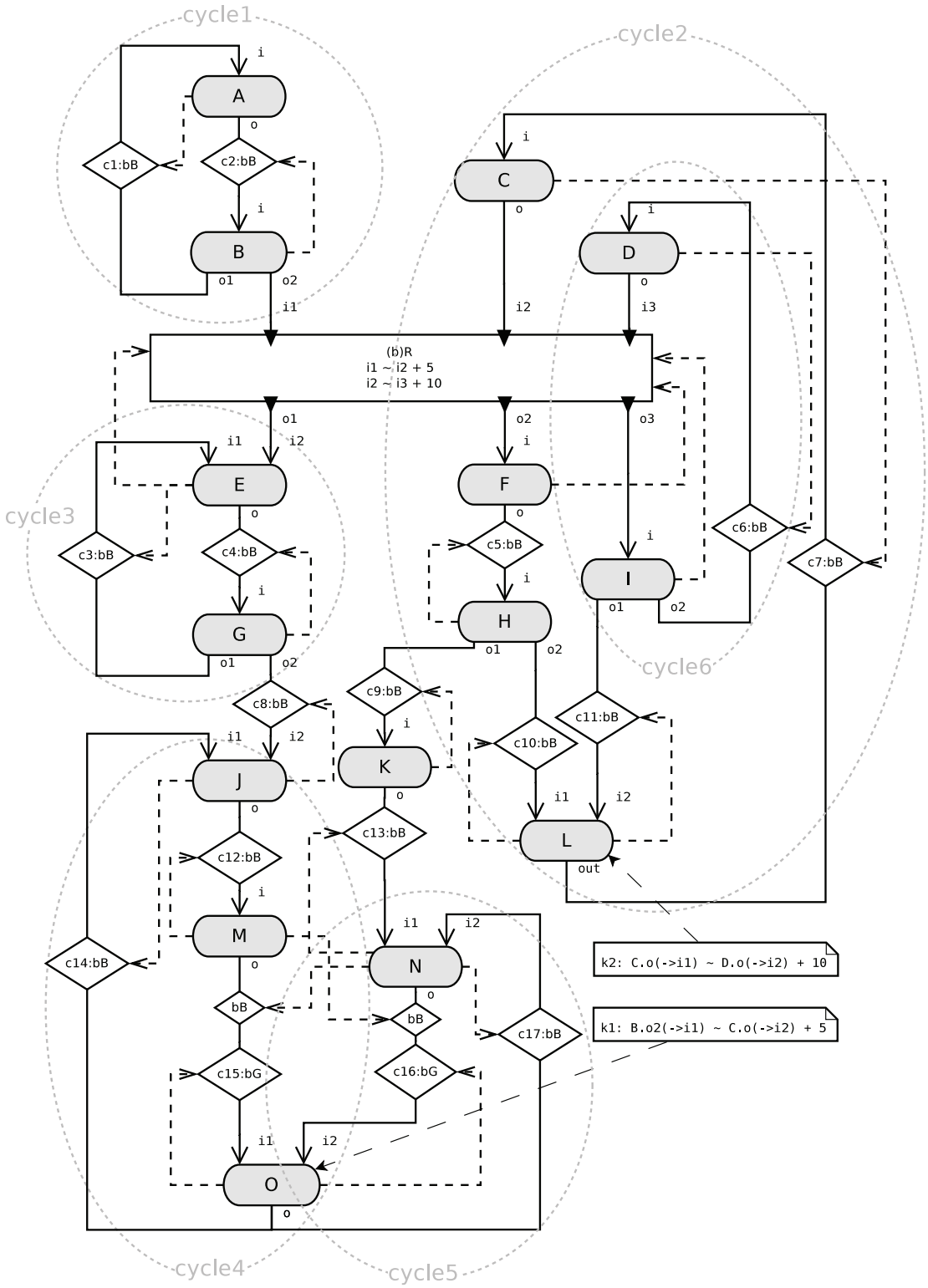


FIGURE 5.13 – Graphe d'application avec 6 cycles synchrones.

connecteurs se trouvant à l'intérieur de segments synchrones synchronisés (définition 20). Dans le cas où un segment synchrone impliqué dans une cohérence fait partie d'un cycle synchrone, l'algorithme cherchera à modifier un connecteur qui ne fait pas partie de ce segment afin de ne pas perturber sa cohérence. C'est, par exemple, le cas du cycle 3 sur la figure 5.13 dont le connecteur c_4 se trouve dans un segment synchronisé. C'est le connecteur c_3 qui sera, alors, choisi pour le changement.

Enfin, les connecteurs indépendants, ne faisant partie d'aucun motif de cohérence, peuvent être changés sans conditions. Ainsi, pour débloquer le cycle 1, le choix entre transformer c_1 ou c_2 est arbitraire. Notons, en outre, qu'à occurrences égales, le changement d'un connecteur indépendant prime sur celui d'un plateau ou d'un régulateur et ce, afin de limiter l'impact sur d'éventuels chemins non concernés.

La détermination des connecteurs bloquants à changer pour résoudre des cycles synchrones est, comme la segmentation, traitée sur l'ensemble du graphe $\mathcal{G}$ dans un système d'équations linéaires. Chaque connecteur indépendant est représenté par une variable et chaque plateau ou régulateur est représenté par une seule variable. Le domaine des variables est $\{0, 1\}$. Pour chaque cycle synchrone, nous imposons que la somme des variables correspondant aux connecteurs qu'il contient soit égale à 1 : $Eq2_{\mathcal{G}} = \bigcup_{P_i \in \mathcal{P}_{\mathcal{G}}} \{Eq_{P_i} = 1\}$ avec $\mathcal{P}$ l'ensemble des cycles synchrones trouvés dans $\mathcal{G}$ et $Eq_{P_i} = \sum_{c \in P_i} v_c$ avec c un connecteur ou un régulateur compris dans P_i ou bien un plateau dont une jonction est comprise dans P_i.

Pour minimiser le nombre de changements de manière générale, nous ajoutons la fonction objectif $Min_{\mathcal{G}} = Minimize(\sum_{c \in \mathcal{L}'} v_c + \sum_{r \in \mathcal{R}'} v_r \times size(O(r)) + \sum_{\pi \in \prod'} v_\pi \times size(\pi))$ telle que $\mathcal{L}', \mathcal{R}'$ et $\prod'$ sont les ensembles, respectivement, des connecteurs indépendants, des régulateurs et des plateaux compris dans des cycles synchrones et $size(O(r))$ et $size(\pi)$ désignent, respectivement, le nombre de canaux du régulateur r et le nombre de jonctions comprises dans le plateau π. Le système linéaire à résoudre au final est $Eq2_{\mathcal{G}} \cup Min_{\mathcal{G}}$.

Sur l'exemple de la figure 5.13, $Eq2_{\mathcal{G}} = \{v_{c_1} + v_{c_2} = 1, v_{c_7} + v_R = 1, v_{c_3} = 1, v_{c_{14}} + v_{c_{15}c_{16}} = 1, v_{c_{15}c_{16}} + v_{c_{17}} = 1, v_{c_6} + v_R = 1\}$ et $Min_{\mathcal{G}} = Minimize(v_{c_1} + v_{c_2} + v_{c_3} + v_{c_6} + v_{c_7} + v_{c_{14}} + v_{c_{17}} + 3 \times v_R + 2 \times v_{c_{15}c_{16}})$. Le résultat de la résolution est : $v_{c_1} = 1, v_{c_2} = 0, v_{c_3} = 1, v_{c_6} = 1, v_{c_7} = 1, v_{c_{14}} = 0, v_{c_{17}} = 0, v_R = 0, v_{c_{15}c_{16}} = 1$.

5.4 Parallélisme

Comme précisé en début de chapitre, l'ensemble des transformations pour générer un graphe d'application, cohérent et dénué d'interblocages, sont effectuées sur le graphe d'application plié. Comme l'indique la section 4.2.1, les composants de notre modèle peuvent être parallèles, précisément selon la logique *Single Program Muliple Data* (*SPMD*) [21]. Selon le modèle d'exécution adopté, deux politiques de distribution des données peuvent être envisagée :

- Une donnée de gros volume est divisée en plusieurs blocs qui seront traités en parallèle. À cet effet, le composant en charge du traitement de la donnée est, lui aussi, répliqué autant de fois que le nombre de blocs ;
- L'ensemble de la donnée est envoyée à chaque instance du composant parallèle.

Un nouvel élément du modèle, appelé *routeur*, est placé en amont et en aval du composant parallèle pour découper ou regrouper les blocs données. Il est représenté par un triangle sur les figures 5.14 et 5.15. Formellement, le routeur est un quadruplet (a, t, I, O) avec :

- a son arité ;

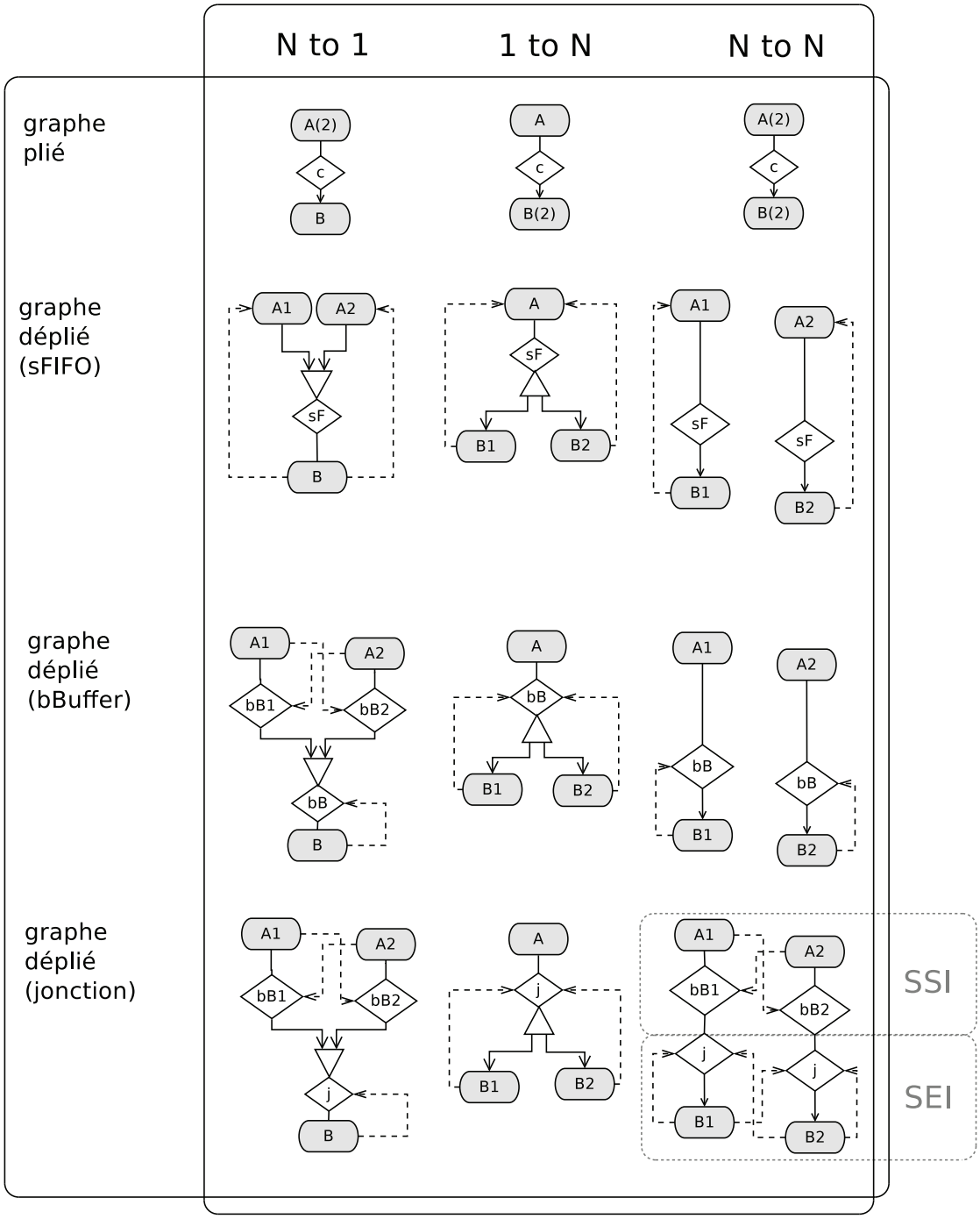


FIGURE 5.14 – La représentation pliée et dépliée des motifs de synchronisation.

- t son type : *scatter* (découpe la donnée en a blocs), *broadcast* (duplique la donnée a fois) ou *gather* (assemble a blocs de données) ;
- I l'ensemble de ses ports d'entrée de données, de taille a s'il est de type gather et de taille 1 sinon ;
- O l'ensemble de ses ports de sortie de données, de taille 1 s'il est de type gather et a sinon.

Le routeur ne possède pas de port de déclenchement. Il agit comme un greffon au connecteur (ou au régulateur) qui distribue ou rassemble les données après émission ou avant réception par ce dernier. Si c'est un gather, dès que chacun de ses ports contient au moins un message, il prélève le plus ancien message de chaque port et assemble ces messages. En cas de scatter, notre modèle de composants étant indépendant de tout type de données, il ne précise donc pas sur quelle base -par exemple un nombre de blocs d'octets sur une donnée sérialisée- le découpage est opéré. Cette décision est déléguée à l'implémentation du connecteur dans le modèle d'exécution concrétisant notre modèle.

Après génération du graphe d'application et sa transformation pour établir des cohérences ou pour débloquer des cycles synchrones, un post-traitement peut générer, à partir du nombre d'instances spécifié par l'utilisateur pour chaque composant, le graphe déplié. Des motifs de composition, classés par arité (N to 1, 1 to N ou N to N) et type de connecteur dans la figure 5.14, sont automatiquement placés entre les instances pour garantir que les blocs de données consommés à chaque itération par les instances d'un composant correspondent bien au même message produit par le composant précédent.

5.4.1 Connexion N to 1

C'est une connexion allant de N instances d'un composant parallèle A vers un composant non parallèle B. Le routeur a la charge de regrouper, à chaque itération de A, les résultats issus de ses instances. Cependant, les instances d'un même composant pouvant fonctionner à différentes fréquences, il est nécessaire de synchroniser leurs messages sortants afin que le routeur ne reçoive que des couples de messages appartenant à la même itération de l'émetteur. Pour ce faire, un bBuffer est ajouté en aval de chaque instance. Il permet de stocker le message que l'instance aura produit le temps que les autres terminent la même itération. Le déclenchement croisé $(A_1.e, bB_2.s)$ et $(A_2.e, bB_1.s)$ assure alors que le message de chaque instance est transmis au reste de l'application lorsque toutes les autres auront fini et donc émis leurs propres messages. Ce motif de composition s'appelle la *sychronisation en sortie d'instances.*

Définition 28 (Synchronization en sortie d'instances) *Dans un graphe d'application $\mathcal{G}$, une synchronisation en sortie d'instances ou SSI est un motif de composition impliquant*

- *deux composants C_1 et C_2,*
- *deux connecteurs bBuffer bB_1 et bB_2 succédant respectivement à C_1 et C_2,*
- *un déclenchement croisé avant constitué de $(C_1.e, bB_2.s)$ et $(B_2.e, bB_1.s)$.*

5.4.2 Connexion 1 to N

C'est une connexion allant d'un composant non parallèle A vers N instances d'un composant parallèle B. Pour être cohérentes, les instances de B doivent être synchronisées afin d'accepter, à chaque itération, des blocs de données issus du même message de A. Le lien de déclenchement qui les relie toutes au port s du connecteur c, du régulateur, ou du composant A si c est de type sFIFO garantit cette synchronisation.

5.4.3 Connexion N to N

C'est une connexion allant de N instances d'un composant parallèle A vers N instances d'un composant parallèle B. La coordination des instances diffère selon le type de connecteur.

Via un connecteur élémentaire

N instances du connecteur c sont nécessaires pour transmettre les données entre instances de même rang. Si les instances de A ne reçoivent que des blocs de données cohérents et simultanés, cette cohérence est maintenue en entrée de B dans les cas où c est de type sFIFO ou bBuffer. En effet, sFIFO et bBuffer conservent l'intégrité et l'ordre des flots de données les traversant.

En revanche, si c est une jonction, une incohérence peut s'introduire si les instances de A ne délivrent pas leurs messages au reste de l'application au même moment. Par exemple, si ces deux instances alimentent des connecteurs c_1 et c_2 non bloquants, il se peut que la lecture de ces connecteurs surviennent alors qu'un seul des deux a été alimenté et un message vide remplacera le contenu de l'autre. Pour éliminer ce risque, nous utilisons le motif de SSI présenté plus haut.

En admettant que les données arrivant aux instances de c soient cohérentes, de manière symétrique à la synchronisation en sortie, il faut s'assurer que ces données soient lues simultanément par les instances de B. À défaut, si c est avec perte, il se peut que B_2 lise le contenu de c_2 et que le contenu de c_1, lui, soit écrasé juste avant sa lecture. Pour prévenir ce genre de situations, nous utilisons un motif de composition appelé la *synchronisation en entrée d'instances.*

Définition 29 (Synchronization en entrée d'instances) *Dans un graphe d'application $\mathcal{G}$, une synchronisation en entrée d'instances ou SEI est un motif de composition impliquant*

- *deux composants C_1 et C_2,*
- *deux connecteurs c_1, c_2 du même type et précédant, respectivement, C_1 et C_2,*
- *un déclanchement croisé arrière constitué de $(C_1.e, c_2.s)$ and $(C_2.e, c_1.s)$.*

Les deux synchronisations SEI et SSI reposent sur les même principes que, respectivement, les motifs SES et SSS, présentées à la section 5.2.

Via un régulateur

Un régulateur est un connecteur N to N par définition. Si les composants émetteur et récepteur sont tous les deux parallèles, ils doivent avoir le même nombre N d'instances. Les ports du régulateur reliant les deux composants sont alors multipliés par N et des règles de filtrage sont ajoutées pour imposer l'égalité des itérations des messages issus des différentes instances du composant émetteur. Cette égalité entraîne l'application de la règle de filtrage sur tous les couples de messages issus d'une instance de l'émetteur. La figure 5.15 illustre ce motif. Dans cet exemple, A et son destinataire C ont deux instances chacun. La règle filtrage est appliquée entre les couples de messages issus de A_1 et de B. À cause de l'égalité imposée automatiquement entre les messages de A_1 et de A_2, la règle de filtrage s'applique, indirectement, également entre les messages de A_2 et de B. Ce principe persiste et est toujours valable si B aussi est parallèle.

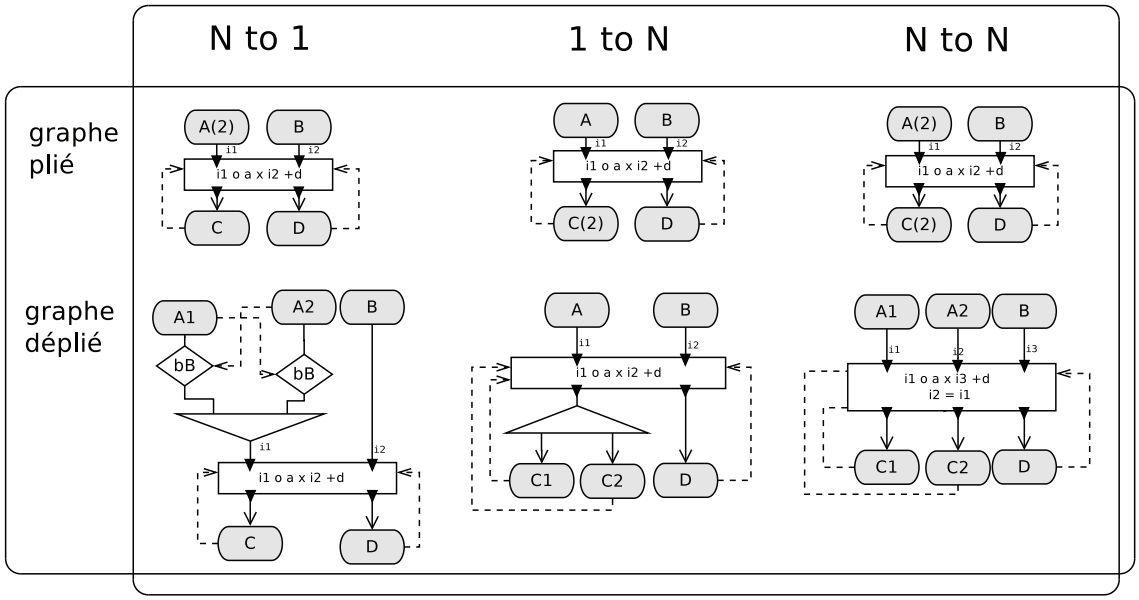


FIGURE 5.15 – Connexion entre composants parallèles via un régulateur.

5.5 Conclusion

Dans ce chapitre, nous avons présenté les procédures permettant de générer automatiquement un graphe d'application répondant à la spécification de l'utilisateur. Grâce à la méthode présentée, nous avons démontré qu'une cohérence temporelle flexible pouvait être obtenue dans un graphe flot de données sans nécessairement n'utiliser que des connexions synchrones. Lorsque des cohérences, strictes ou non, sont requises, les motifs de composition placés tendent à conserver un maximum de la désynchronisation initiale de l'application et donc, en d'autres termes, de sa performance. Comme le laissent paraître certains exemples du chapitre, à partir d'une certaine taille de graphe, la mise en place manuelle de ces motifs devient très compliquée, même pour une personne familière de notre modèle de composition. Notre approche consistant à générer un graphe d'application à partir de propriétés vise l'assister dans cette opération.

Nous avons également présenté une méthode pour détecter et résoudre les interblocages dans les graphes d'application sans nuire aux cohérences déjà établies dedans. Toutes les opérations décrites dans ce chapitre concernent le graphe d'application plié. Au cas où certains composants de l'application seraient parallèles, les règles de dépliage données à la section 5.4 permettent, ensuite, d'en déduire sa version dépliée.

Dans la partie suivante de ce mémoire, nous aborderons la mise en pratique de notre modèle de composants et de toutes les méthodes décrites précédemment. Nous présenterons un modèle d'exécution basé sur l'intergiciel FlowVR et nous valoriserons notre approche de construction automatique à travers deux applications réelles.

Troisième partie

Résultats expérimentaux

Chapitre 6

Implémentation

Sommaire

6.1 L'intergiciel FlowVR

Nous utilisons l'intergiciel FlowVR [1] comme modèle d'exécution. Il permet de créer et d'exécuter des applications C++ haute-performance distribuées sur systèmes Unix. Dans cette section, nous présentons succinctement son modèle de composants et les principes de son utilisation.

6.1.1 Le modèle FlowVR

Le modèle FlowVR comprend trois objets de base, les *modules* et les *filtres*, assemblés au sein d'un *réseau* à travers des *connexions*.

Les modules

Les composants d'une application FlowVR sont appelés "modules" et ont la même sémantique que les composants de notre modèle à deux différences près :

1. Leur port de déclenchement, s'il est connecté, est bloquant dès la première itération,
2. Tous leurs ports, hormis le port de signalement, ne peuvent être connectés qu'à un seul lien chacun.

Les ports de déclenchement et de signalement dans un module FlowVR s'appellent, respectivement, *beginIt* et *endIt*. Les modules apparaissent sous la forme de rectangles verts aux coins arrondis sur la figure 6.1.

Les filtres

L'environnement FlowVR fournit également des filtres qui s'utilisent entre deux modules. Ils permettent d'établir toutes sortes de politiques de communication : découper ou fusionner des messages, fixer une fréquence ou un ordre de passage, etc. Selon leurs fonctions, certains peuvent avoir plus d'un port d'entrée ou de sortie. Chaque filtre possède un port de déclenchement appelé *order*. Comme le port *beginIt* du composant, il ne peut être connecté qu'à un seul lien de déclenchement.

Les filtres peuvent accéder aux estampilles des messages pour effectuer leurs filtrages. FlowVR définit, à cet effet, un type particulier de filtres, appelés *synchroniseurs*, qui ne traitent que les estampilles. L'assemblage de synchroniseurs et de filtres classiques permet de créer des objets de coordination complexes.

Le filtre est représenté par un losange bleu sur la figure 6.1 et le synchroniseur par un rectangle rouge. Comme les modules, ils sont tous deux bloquants à leur première itération sur tous leurs ports. Pour prévenir les interblocages, il existe un filtre appelé *PreSignal*. Son rôle est d'émettre un message vide au démarrage puis, de simplement relayer tous les messages qui lui arrivent par la suite. D'autres filtres souvent utilisés dans FlowVR sont les suivants :

FilterIt Ce filtre possède deux ports d'entrée dont un uniquement destiné à recevoir des estampilles. Il empile tous les messages entrants et délivre le plus ancien à chaque fois qu'il reçoit un message sur son port *order*. Par défaut ce filtre est non bloquant : faute de nouveaux messages, il réémet le dernier message émis. Un paramètre permet toutefois de le rendre bloquant.

RoutingNode Il fait partie des filtres servant à la distribution des données. Il duplique un message entrant en autant de copies qu'il a de ports de sortie connectés.

Scatter Il sert aussi à la distribution de données. Plutôt que de dupliquer le message comme le RoutingNode, il le découpe en autant de morceaux de taille égale qu'il a de ports de sortie connectés.

Merge Ce filtre réalise l'opération inverse au Scatter. Il concatène en un seul message les blocs de données reçus par tous ses ports d'entrée.

SignalAnd C'est un synchroniseur qui transmet l'estampille reçue sur son premier port d'entrée dès que tous ses autres ports d'entrée sont alimentés. Cela peut servir à effectuer un ET logique de plusieurs sources.

Les connexions

Les filtres ne sont pas indispensables dans une application FlowVR car le vecteur de communication est la connexion. La connexion est un canal de communication FIFO reliant un port de sortie à un port d'entrée. Les connexions transportent des messages. Ceux-ci peuvent contenir des données et des estampilles, des données seules ou des estampilles

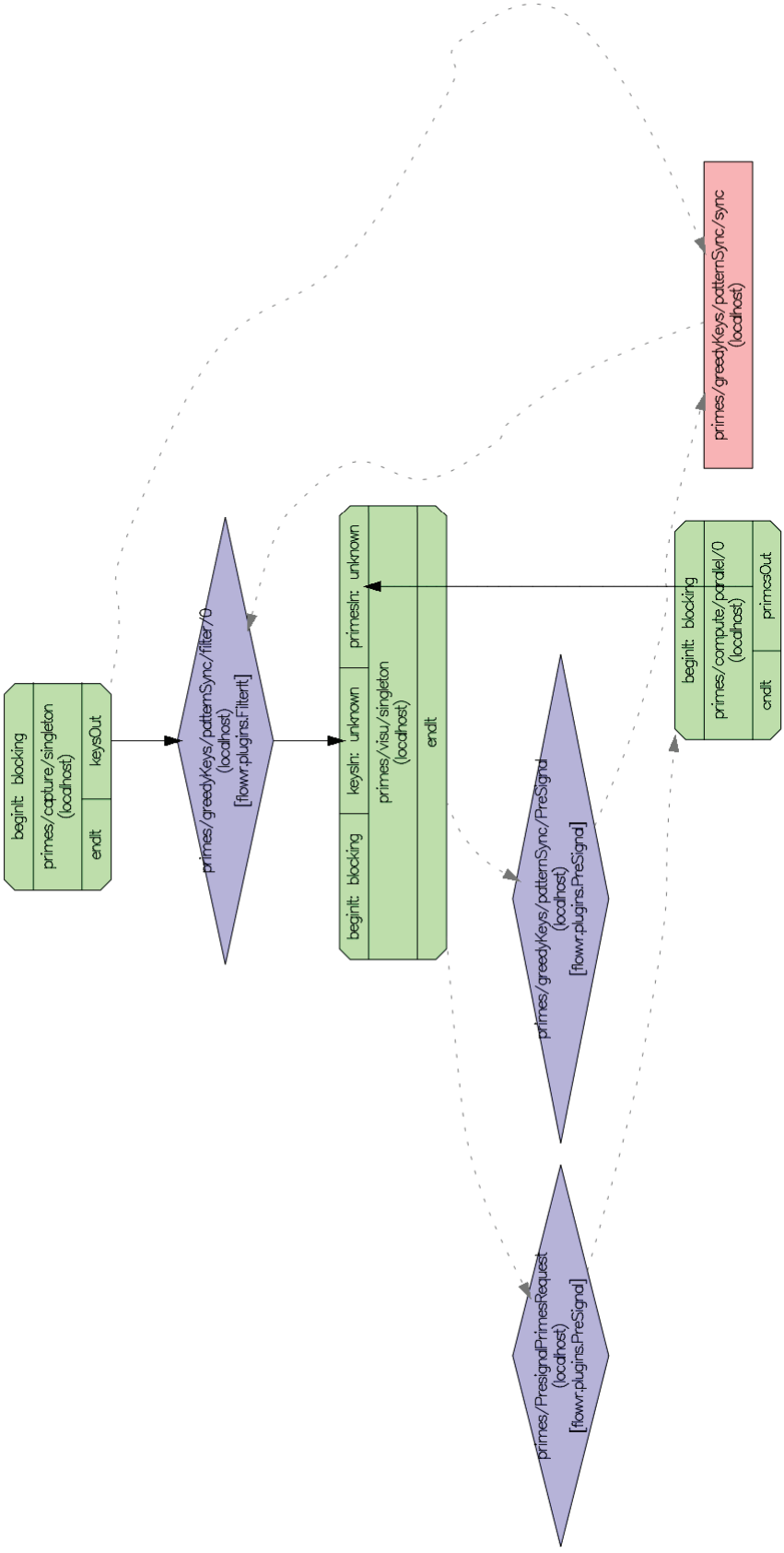


FIGURE 6.1 – Réseau FlowVR de l'application Primes.

seules. Dans les deux premiers cas, l'objet se nomme *Connection*. Dans le dernier, il porte le nom de *ConnectionStamps*. Il est alors généralement relié à un filtre, un synchroniseur ou peut servir à déclencher un module. Étant de simples canaux FIFO, les connexions ne protègent pas leurs récepteurs contre la saturation ou le blocage. L'usage d'un filtre est alors nécessaire.

Sur la figure 6.1, les connexions ne transportant que des estampilles sont représentées par des flèches en pointillés. Les connexions pouvant transporter des données sont représentées par des flèches en trait plein.

L'application FlowVR

Une application FlowVR est représentée par un réseau de modules et de filtres reliés par des connexions. Un réseau peut être encapsulé à l'intérieur d'un composant grâce à l'architecture hiérarchique [53] du modèle FlowVR. On parle alors de module *composite*.

Au sein du réseau, les modules sont considérés comme des boîtes noires et leurs ports des points d'entrée et de sortie de données. En conséquence, un module n'a pas connaissance des autres modules. À l'exécution, les modules ne s'échangent pas de messages directement. Au lieu de cela, les messages issus d'un module transitent par un processus local à sa machine hôte, appelé *démon FlowVR* [1], qui gère un segment de mémoire partagée sur cette machine. Les filtres, eux, sont implémentés comme des plugins pour les démons. Le système FlowVR fait en sorte que les démons des différents hôtes s'échangent les données.

La figure 6.1 illustre *Primes*, une application simple fournie en exemple avec la distribution FlowVR. Elle est constituée de trois modules :

- *compute* : module qui calcule une suite de nombres premiers ;
- *visu* : module qui dessine, à l'écran, une spirale formée de tous les nombres au fur et à mesure qu'ils sont générés ;
- *capture* : module qui détecte la pression sur les flèches du clavier pour changer l'angle de vue dans *visu*.

compute et *visu* sont reliés par une connexion FIFO. Pour éviter d'être saturé par *compute*, une autre connexion permet à *visu* de le déclencher. Pour éviter l'interblocage, un filtre PreSignal est placé sur cette connexion. La communication entre *capture* et *compute* est, elle, contrôlée par l'assemblage d'un filtre FilterIt, d'un synchroniseur Greedy et d'un filtre PreSignal. Avec cet assemblage, le filtre ne transmet un message que sur ordre du synchroniseur et ce dernier n'émet un ordre que lorsque *visu* le déclenche, le protégeant, ainsi, de la saturation.

La figure 6.2 représente le réseau de l'application Primes lorsque le module *compute* est *parallélisé*. Un filtre Merge est alors placé à la sortie des deux instances de ce composant.

6.1.2 La programmation FlowVR

La programmation d'une application FlowVR passe par l'implémentation de ses modules puis par la composition de l'application elle-même. L'utilisateur dispose d'une interface de programmation (*Application Programming Interface* ou *API*, en anglais) en C++ pour réaliser ces deux étapes. Plusieurs dizaines de filtres sont fournis dans la distribution de FlowVR pour combler un maximum de besoins. Un utilisateur avancé peut également programmer les siens.

Programmer un composant

Programmer un composant avec l'API de FlowVR consiste à :

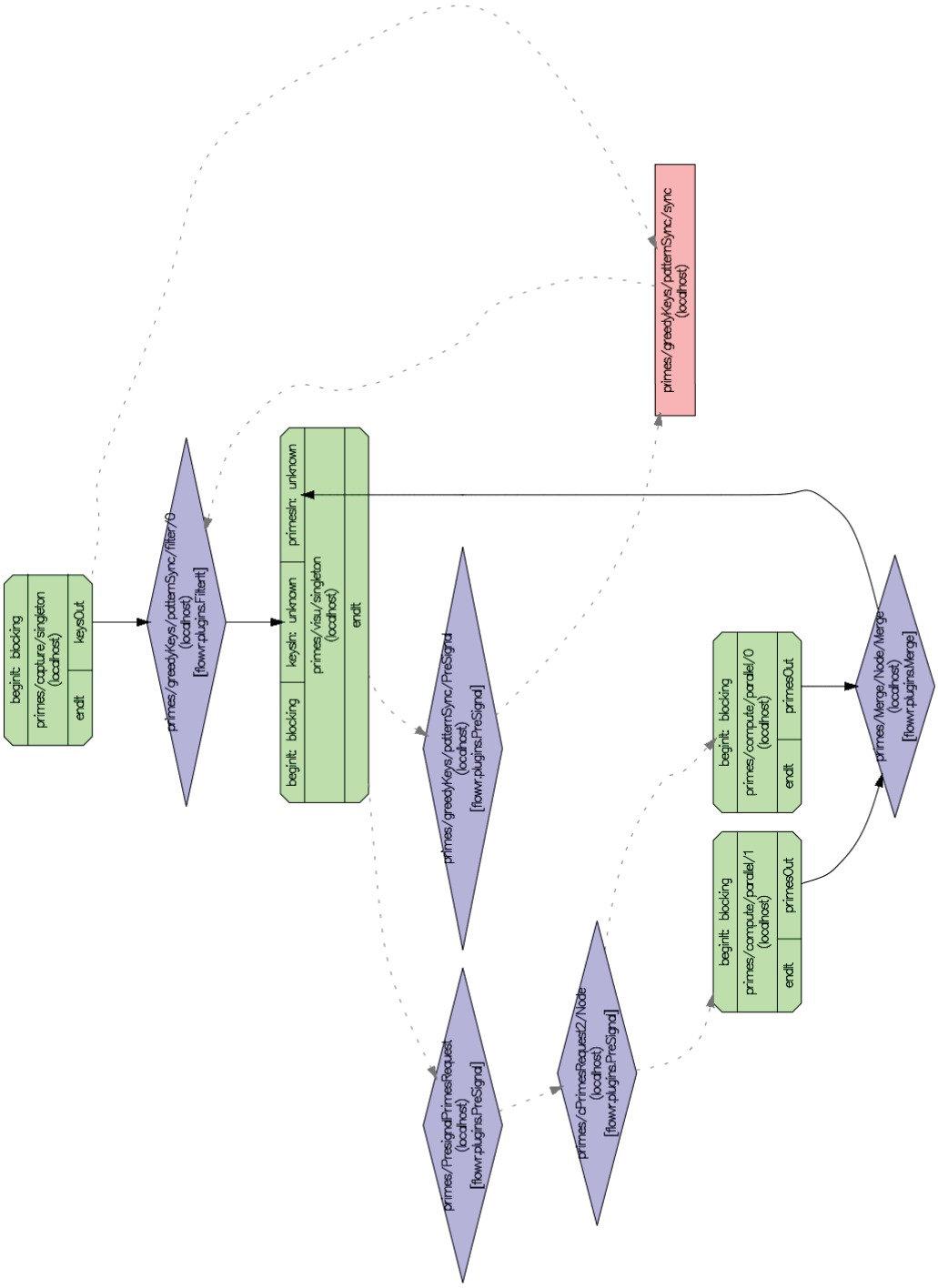


FIGURE 6.2 – Réseau FlowVR de l'application Primes où le module *compute* est parallélisé.

1. Déclarer un ensemble de ports et les estampilles qui leur sont associées ;
2. Déclarer un objet Module et l'initialiser ;
3. Déclarer des variables Messages et leur allouer un espace en mémoire ;
4. Insérer le code de calcul à l'intérieur d'une boucle "wait-get-put".

En plus de son implémentation, l'utilisateur doit créer, pour chaque module, deux fichiers d'en-tête C++ qui servent à le décrire dans les applications auxquelles il sera intégré :

- Le fichier module*.h [1] sert à renseigner les ports du module, ses éventuels paramètres et donner une description textuelle de sa fonction ;
- Le fichier metamodule*.h indique les conditions de lancement du module : instance unique ou MPI et sa commande de lancement.

Programmer une application

La composition de l'application s'effectue dans un fichier *.comp.cpp [2] où l'utilisateur instancie modules, filtres, synchroniseurs et connexions. Du fait de l'approche hiérarchique du modèle de FlowVR, ce fichier doit également être accompagné d'un en-tête *.comp.h, équivalent au module*.h.

La compilation de modules et d'applications se fait via l'outil CMake [63]. L'utilisateur doit donc également écrire les fichiers de compilation nécessaires. Chaque module ou application est ensuite compilé en fichier binaire.

L'utilisateur peut lancer l'application sur une seule machine ou sur une architecture distribuée. Dans le second cas, il doit alors fournir la carte de déploiement de l'application sous la forme d'un fichier .csv. De plus, tous les paramètres de l'application, y compris ceux définis par l'utilisateur lors de l'implémentation, peuvent être renseignés à son lancement dans un fichier texte fourni en paramètre. Cela permet de modifier le déploiement de l'application ou ses données d'entrée sans avoir à la recompiler.

Au démarrage de l'application, FlowVR vérifie la consistance du réseau puis regroupe, dans 4 fichiers .xml, toutes les modalités d'exécution : description du réseau, commandes de lancement avec options et adresses IP des différents hôtes.

FlowVR convient parfaitement à la visualisation scientifique interactive en termes de performances. Cependant, la programmation -et surtout la coordination- d'applications peut-être un travail long auquel l'utilisateur se prend parfois à plusieurs reprises avant de parvenir à un résultat satisfaisant. En concevant notre modèle de composants, nous avons cherché à alléger cette difficulté en proposant une couche d'abstraction supérieure pour la construction d'applications qui est le graphe de spécification.

La transposition de notre modèle décrit au chapitre 4 dans le modèle FlowVR sera présentée à la section 6.3. Nous avons également implémenté la méthode de déduction automatique de graphes d'application décrite au chapitre 5. Nous prouvons ses avantages sur des exemples réels à la section 6.4. Enfin, nous avons aussi développé une surcouche à l'API FlowVR afin de faciliter la transformation de codes sources en composants FlowVR. Nous la présentons ci-dessous.

1. * est généralement remplacé par le nom du module.
2. * est généralement remplacé par le nom de l'application.

6.2 FVmoduleAPI

Pour intégrer au sein d'une application un programme existant, nous avons privilégié l'annotation à d'autres solutions, principalement à base de manipulation de binaire à la volée [85]. Ceci est dû au surcoût qu'induisent ces dernières. L'annotation a aussi l'avantage de laisser l'utilisateur placer ses points d'entrée et de sortie dans le code cible avec plus de précision.

L'utilisateur final peut effectuer l'annotation lui-même car il connaît bien le code en question. Cependant, un écueil courant des API de couplage est leur relative complexité pour un non informaticien. L'API de FlowVR, conçue pour programmer des composants à partir de zéro, n'est pas adaptée à cet usage. Nous lui avons alors créé une surcouche appelée FVmoduleAPI [55]. Elle masque à l'intérieur de fonctions de haut niveau tous les détails de l'API d'origine. Elle consiste en une série d'instructions à insérer dans le code afin qu'il sache recevoir, interpréter et envoyer des messages FlowVR. Les objectifs de FVmoduleAPI sont :

1. d'être aussi flexible que l'API FlowVR en terme de nombre de ports possibles, de types de données, de points d'envoi et de réception de données dans le code ;
2. de limiter au maximum la quantité de code à ajouter au programme cible ;
3. d'être compatible avec C et Fortran en plus de C++.

L'utilisation de l'API FlowVR comporte beaucoup d'étapes répétitives comme la création d'objets Module et Port au début du programme ou encore celle d'objets Message à l'envoi et à la réception de données. Nous avons donc cherché, en premier lieu, à automatiser la gestion de ces variables afin d'éviter à l'utilisateur de les manipuler. L'une des caractéristiques majeures de FVmoduleAPI est aussi qu'elle permet de créer des composants en C et en Fortran, en plus du langage d'origine de l'API FlowVR qui est le C++. Cela est rendu possible grâce à la conception "tout fonction" de la librairie. Ainsi, bien qu'elle soit écrite en C++ -car utilisant des classes FlowVR, son interface consiste en des fonctions ayant des arguments de types élémentaires (int, char*, void*, etc). À partir de là, il est facile de rendre ces fonctions accessibles dans du code C grâce à la directive `EXTERN "C"`. La compatibilité avec Fortran devient alors possible. En effet, les compilateurs Fortran les plus récents (comme gfortran) [59] peuvent lier des programmes Fortran à des fonctions C au prix de légers changements aux en-têtes de ces fonctions pour l'adapter aux conventions de nommage de Fortran. Deux autres contraintes sont également à respecter dans la spécification de fonctions Fortran. La première est que tous les arguments sont passés par adresse. La seconde est que tous les arguments de type chaîne de caractères doivent être accompagnés d'un autre argument entier indiquant leur longueur. La valeur de cet argument n'a pas à être renseignée par l'utilisateur, elle est récupérée automatiquement à l'appel de la fonction.

En prenant toutes ces particularités en compte, nous avons défini, pour chaque fonction de FVmoduleAPI, une fonction d'enveloppement spécifique de manière à ce que l'API Fortran soit identique à celle pour C et C++. FVmoduleAPI propose 15 fonctions et procédures dont 8 suffisent à transformer un code en composant FlowVR. Les appels aux procédures `fv_addinputport(char* name)` et `fv_addoutport(char* name)` permettent d'ajouter autant de ports d'entrée et de ports de sortie que souhaité avant d'initialiser le composant par un appel à `fv_initmodule(int nbnodes, int rank)`. L'utilisateur doit ensuite encapsuler le code de calcul à l'intérieur d'une boucle dont la condition de poursuite est le résultat de la fonction `fv_moduleruns()`. Comme pour nombre de programmes de simulation, si le code est déjà itératif par nature, le résultat de `fv_moduleruns()` est

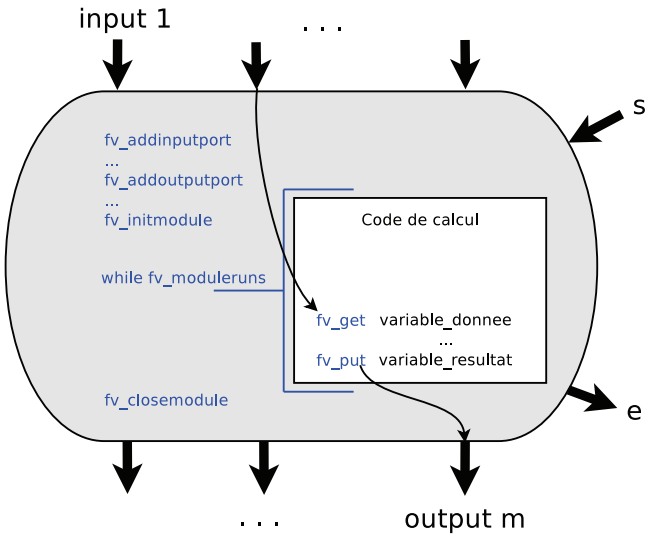


FIGURE 6.3 – Intégration d'un code de calcul dans un composant à l'aide de FVmoduleAPI.

ajouté à la condition de poursuite de cette boucle. Pour que le composant s'arrête correctement en même temps que l'application, il est nécessaire d'ajouter aussi l'instruction `fv_closemodule()` juste après la boucle. On peut ensuite choisir, au milieu de ce code, les points de réception et d'envoi de données en plaçant des appels à `fv_get( void* data, char* port)` et `fv_put(const void* data, int size, char* port)`.

La gestion de la mémoire dans FVmoduleAPI demande un minimum d'attention. Pour rester générique par rapport aux types de données, l'API, comme FlowVR, traite les variables lues par `fv_put` ou affectées par `fv_get` comme des données brutes sérialisées. Ainsi, pour émettre des données, `fv_put` demande, dans ses paramètres, la taille en mémoire de la variable pour pouvoir la transformer en message FlowVR. En réception, l'utilisateur doit s'assurer que la variable de destination convient à la donnée reçue car aucune conversion n'est effectuée implicitement. Si la taille de la donnée varie, la fonction `fv_getdatasize(const char* port)` permet de la connaître et de réallouer suffisamment de mémoire à la variable.

La figure 6.3 aide à situer ces ajouts au sein du code cible. Les autres procédures et fonctions de la librairie permettant de gérer le parallélisme du composant ou bien d'associer aux ports des estampilles supplémentaires -l'estampille d'itération existe par défaut, de les lire et d'écrire dedans. La liste des fonctions de FVmoduleAPI est fournie annexe A.

6.3 Transposition de notre modèle dans FlowVR

Dans cette section, nous expliquons comment nous avons représenté notre modèle de composants au sein de FlowVR. Pour accomplir totalement cette transposition, nous avons dû créer quelques nouveaux filtres. Le tableau 6.1 donne un aperçu des solutions que nous avons adoptées pour implémenter chaque élément de notre modèle.

6.3.1 Le composant

Le module est l'objet FlowVR légitime pour représenter le composant de notre modèle. Cependant, comme évoqué dans la section 6.1.1, son port de déclenchement est bloquant

Element	**Solution avec FlowVR**
Composant	Module + PreSignal + SignalAnd + RoutingNode
sFIFO	Connection + ConnectionStamps
bBuffer	Nouveau filtre + SignalAnd + RoutingNode
nbBuffer	Nouveau filtre + SignalAnd + RoutingNode
bGreedy	Nouveau filtre + SignalAnd + RoutingNode
nbGreedy	Nouveau filtre + SignalAnd + RoutingNode
Régulateur	Nouveau filtre + SignalAnd + RoutingNode
Lien de données	Connection
Lien de déclenchement	ConnectionStamps
Routeur "broadcast"	RoutingNode
Routeur "scatter"	Scatter
Routeur "gather"	Merge + SignalAnd

TABLE 6.1 – Choix d'implémentation de notre modèle de composants.

dès la première itération. Il est donc nécessaire de lui attacher un filtre PreSignal pour le rendre non-bloquant à cette première itération. De plus, ce port ne peut recevoir qu'un seul lien de déclenchement. Or, dans notre modèle, un composant peut, par exemple, alimenter deux autres composants par le biais de connecteurs sFIFO. Il devra alors être déclenché par eux deux simultanément. Il est donc nécessaire de placer un filtre SignalAnd en amont du PreSignal afin de récolter les signaux des différents récepteurs. Enfin, toujours d'après la section 6.1.1, les ports de sortie de données du module ne peuvent, eux aussi, accueillir qu'un seul lien. Pour desservir plusieurs récepteurs comme le permet notre spécification du composant, il faut placer un filtre RoutingNode en sortie de chacun de ces ports.

La figure 6.4 illustre cet assemblage concrétisant notre spécification du composant.

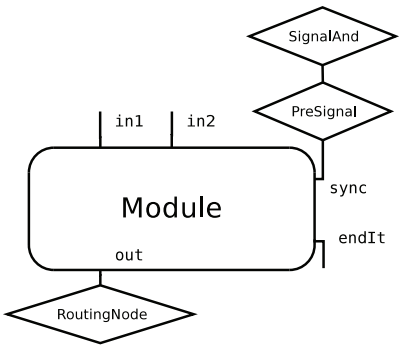


FIGURE 6.4 – Représentation du composant dans le modèle FlowVR.

6.3.2 Les connecteurs

Le connecteur sFIFO de notre modèle est facilement représentable dans FlowVR. Il consiste, en effet en :

- Une *Connection* allant d'un port de sortie du composant émetteur à un port d'entrée du composant récepteur,
- Une *ConnectionStamps* allant du port endIt du composant récepteur au port beginIt du composant émetteur.

À une exception près (le filtre *LastOrNull*, semblable au nbGreedy), nous n'avons pas été en mesure de représenter les autres connecteurs grâce aux filtres préexistants dans FlowVR. Pour des raisons d'homogénéité et pour faciliter leur débogage, nous avons choisi d'implémenter les connecteurs bBuffer, nbBuffer, bGreedy et nbGreedy sur la base de deux nouveaux filtres, l'un sans perte et l'autre ne stockant que le dernier message reçu. Le comportement non bloquant, tel que décrit par notre modèle, est activé ou désactivé dans ces filtres au moyen d'un paramètre.

Le régulateur a également été implémenté en tant que nouveau filtre. Il a la particularité de posséder un unique port d'entrée et un unique port de sortie. Tous deux sont "virtuels", ils peuvent recevoir plusieurs connexions. À la compilation de l'application, un port d'entrée (de sortie) est créé, à l'intérieur du régulateur, pour chaque connexion entrante (sortante). Des canaux de données sont ensuite établis entre chaque port d'entrée et le port de sortie de même rang. Ce système s'inspire du filtre *Merge* de FlowVR qui a une seule sortie et un nombre indéfini d'entrées. Le filtrage des messages en fonction de leurs numéros d'itération est réalisé à l'intérieur du régulateur par la librairie de résolution de systèmes linéaires lp_solve [6].

6.3.3 Le parallélisme

Les filtres RoutingNode, Scatter et Merge de FlowVR correspondent, respectivement, aux types de routeur *broadcast*, *scatter* et *gather* définis à la section 5.4. Cependant, comme tout filtre de FlowVR, ils doivent être déclenchés pour fonctionner. Ce déclenchement est opéré par le module émetteur ou, dans le cas d'un filtre Merge, par toutes les instances de celui-ci à travers un filtre SignalAnd. Dans le cas d'une connexion N to 1, ce mécanisme se substitue à la synchronisation en sortie d'instances décrite à la section 5.4.1.

La connexion N to N est déjà implémentée dans FlowVR sous la forme d'un objet abstrait appelé *PatternParallel*. Il ne contient pas de filtre mais cache un motif de composition permettant de relier, par les mêmes ports, les instances de même rang de deux composants. Ce motif ne permet donc que des connexions de type FIFO. Nous lui avons alors développé une extension capable d'accueillir les filtres élémentaires que nous avons développés et d'automatiquement mettre en place les mécanismes de SSI et de SEI visibles sur le dernier schéma de la figure 5.14.

Ces différents objets, lorsqu'employés dans le fichier de composition *.comp.cpp, dictent à l'intergiciel FlowVR la forme dépliée de l'application qui sera générée à son exécution.

6.4 Implémentation de la construction automatique

Nous avons implémenté en C++ la méthodologie du chapitre 5 qui permet de transformer un graphe de spécification en un graphe d'application cohérent et sans interblocages.

Le graphe de spécification doit être fourni, en entrée du programme, sous la forme d'un fichier XML. L'annexe E donne un exemple d'un tel fichier. Dans ce fichier, l'utilisateur renseigne, pour chaque composant, son nom, ses ports, s'il est interactif et le nombre de ses instances s'il doit être parallélisé. Au niveau de chaque composant, l'utilisateur peut, à la suite des ports, définir des contraintes de cohérence. Dans chacune, il associe, à deux reprises, un port d'entrée à un port de sortie d'un autre composant. Il précise aussi l'opérateur et les constantes de la formule de cohérence. La deuxième partie du fichier concerne les connexions. Pour chacune l'utilisateur renseigne les composants et ports source et destination ainsi que les politiques de perte et de blocage.

Le programme extrait les informations stockées dans ce fichier afin de construire un graphe de spécification sur lequel sont appliquées, dans le même ordre, les transformations décrites au chapitre 5. En particulier, le problème d'égalisation du nombre de jonctions -nommé "segmentation" dans la section 5.2- est résolu grâce à la librairie lp_solve, déjà utilisée à l'intérieur des régulateurs.

Le graphe d'application résultant est produit par le prototype au format graphviz [35]. En l'état, une retranscription manuelle dans l'API FlowVR est donc nécessaire pour implémenter dans FlowVR l'application obtenue. La génération directe des fichiers d'application .comp.cpp et .comp.h (voir la section 6.1.2) serait, cependant, une opération triviale et fera partie des suites immédiates de ce travail.

6.5 Études de cas

Dans cette section, nous présentons les résultats de l'application de nos motifs de coordination sur de petits exemples génériques.

6.5.1 Exemples abstraits

Dispositif expérimental

Les exemples d'application ont été implémentés avec l'API FlowVR selon la manière décrite à la section 6.3. Les modules relèvent tous d'un module générique factice que nous avons créé pour l'occasion. Ce module ne réalise aucun calcul. Il itère simplement selon une vitesse fixe ou bien variable à l'intérieur d'un intervalle qui lui est fourni en paramètre. Pour chaque module, différents intervalles de fréquences entre 10 et 100 it/s ont été testés et ce, afin de solliciter aussi bien le comportement avec perte des connecteurs nbGreedy que leur caractère non bloquant.

Les tests ont été réalisés sur la grappe Mirev du Laboratoire d'Informatique Fondamentale d'Orléans et, en particulier, sur ses nœuds 21 à 29. Chaque nœud est une station 2 x QuadCore AMD Opteron 2376 2,3 Ghz avec 16 GB de mémoire RAM.

Les réseaux FlowVR des applications-exemples présentées ci-dessous sont illustrés dans les annexes B, C et D respectivement.

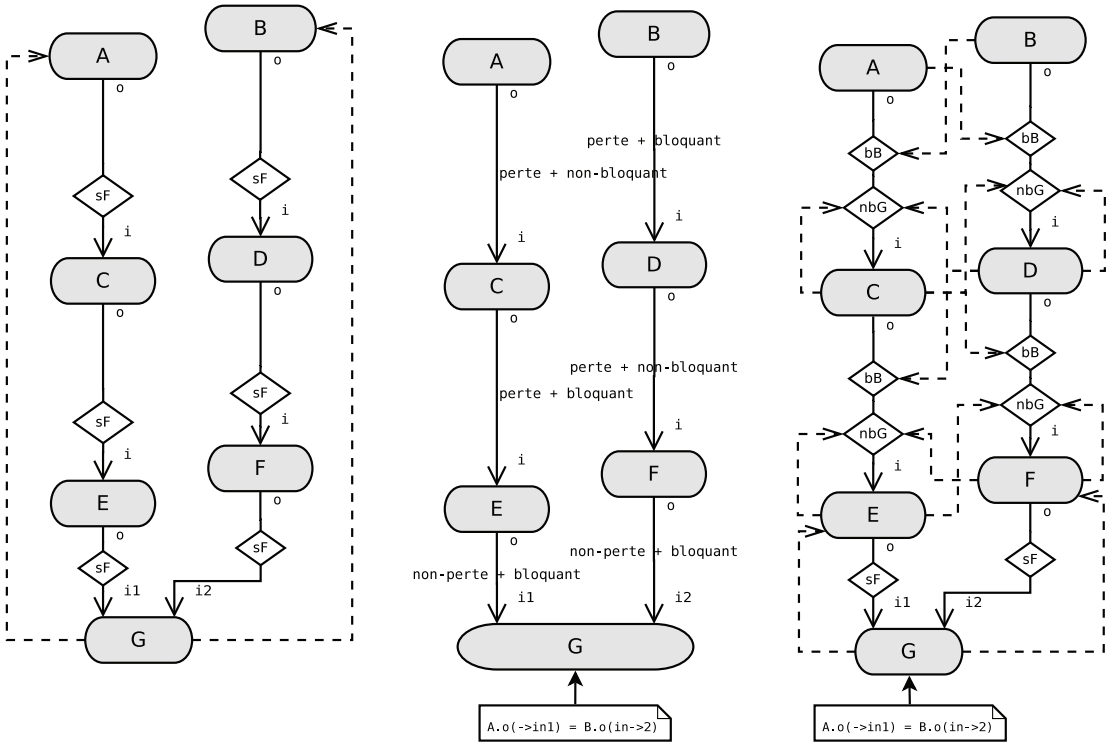
Test de la cohérence stricte

Le premier exemple vise à évaluer l'impact de nos mécanismes de synchronisation sur la performance d'une application lorsqu'une cohérence stricte est requise. Cette application, illustrée par la figure 6.5c, est obtenue à partir de la spécification sur la figure 6.5b. Elle comporte comporte :

- Sept modules à fréquence réglable ;
- Un motif de SSS+SES pour établir une cohérence stricte entre les couples de messages issus des modules A et B ;
- Un motif de SSS+SES entre les couples de modules $\{C, D\}$ et $\{E, F\}$ pour maintenir cette cohérence jusqu'au module G.

Nous avons comparé les performances de cette application à une alternative triviale, illustrée par la figure 6.5a, dans laquelle on synchronise l'ensemble des modules pour obtenir la même cohérence. Dans cette version, tous les connecteurs sont synchrones. Aucun module ne peut donc fonctionner plus rapidement qu'un module qui l'alimente directement ou indirectement. De plus, les liens de déclenchement allant du module G aux modules A et B font que ceux-ci n'itèrent à nouveau que lorsque G a fini son itération. Cette construction

aboutit à un fonctionnement séquentiel de l'application qui garantit la cohérence stricte des messages reçus par G par rapport à ceux issus de A et de B. Toutefois, elle entraîne également la chute des fréquences de tous les modules à celle du module le plus lent.



(a) Une application entièrement synchronisée

(b) Spécification d'une cohérence stricte

(c) Application de la cohérence stricte

FIGURE 6.5 – Un exemple d'application soumise à une cohérence stricte.

En comparaison, l'application utilisant nos motifs de synchronisation demeure relativement asynchrone puisqu'elle comporte 4 zones désynchronisées : {A}, {B}, {C,D} et {E,F,G}. La figure 6.6 illustre cette différence sur une des configurations testées.

Le taux de cohérence des messages en sortie d'un motif de SSS+SES se situe entre 98,83% et 99,67%. Ce résultat a été calculé à partir d'échantillons d'environ 2000 couples de messages non nuls reçus en même temps par E et F. On retrouve cette erreur lors d'une exécution locale de l'application. La validité de nos motifs ayant été démontrée formellement, nous attribuons ce taux d'erreur aux mécanismes de synchronisation internes de FlowVR et aux quelques objets de communication intermédiaires.

Test de la cohérence non stricte

Nous présentons ici deux exemples élémentaires de graphes soumis à des contraintes de cohérence non strictes. L'application de la figure 6.7b est obtenue à partir de la spécification sur la figure 6.7a. Elle comporte :

- Sept modules à fréquence réglable ;
- Un régulateur pour établir une cohérence non stricte entre les couples de messages issus de A et de B ;
- Deux motifs de SSS+SES suivant les couples de modules {C,D} et {E,F} pour maintenir cette cohérence jusqu'au module G.

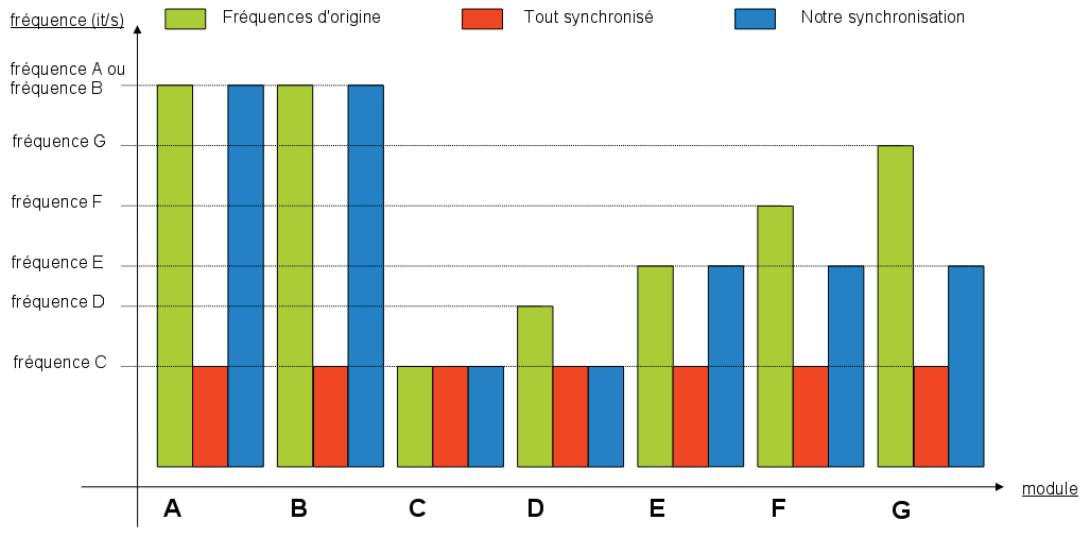


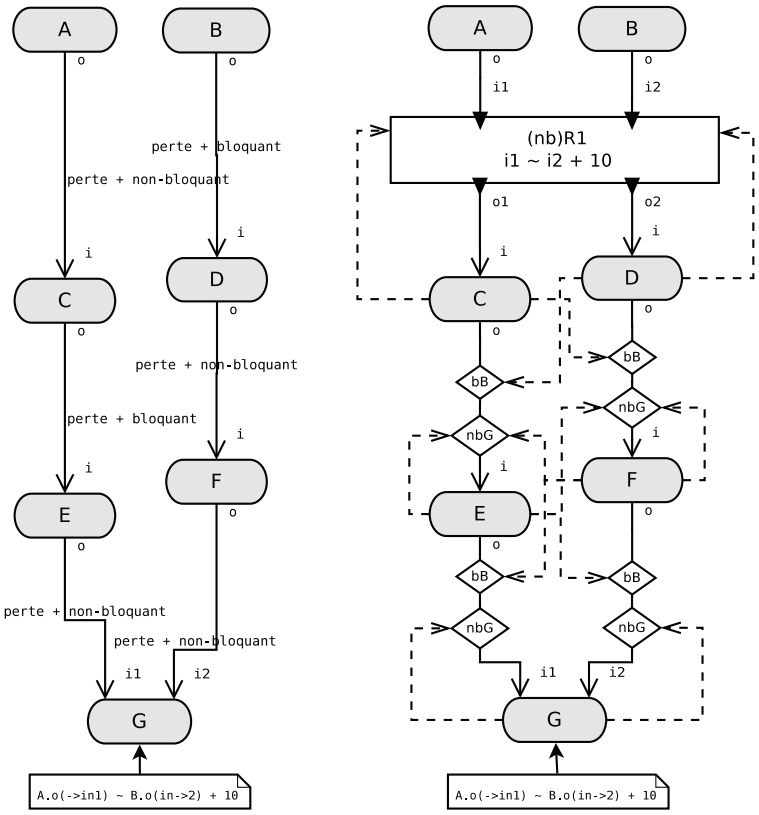
FIGURE 6.6 – Comparatif de performances entre trois types de synchronisation.

Dans la figure 6.8b, l'application est soumise à deux contraintes de cohérence non strictes liées. Ces deux contraintes sont indiquées sur sa spécification, figure 6.8a. Elle comporte :

- Dix modules à fréquence réglable ;
- Un régulateur pour établir une cohérence non stricte entre les couples de messages issus des modules A et B ;
- Un régulateur entre les groupes de modules {C,D,H} et $\{E, F, I\}$ pour maintenir la cohérence précédente jusqu'au modules E et F et en établir une nouvelle entre les couples de messages issus de D et de H ;
- Un motif de SSS+SES suivant le couples de modules {E,F} afin de maintenir la première cohérence jusqu'au module G ;
- Un motif de SSS+SES suivant le couples de modules {E,I} afin de maintenir la première cohérence jusqu'au module J.

Les cohérences souhaitées dans ces deux exemples étant non strictes, elles ne peuvent être accomplies sans un régulateur. Contrairement à un motif de synchronisation SSS+SES, le filtrage dans un régulateur se fait au sein du même objet. Il n'est alors pas tributaire de temps de transmission de messages au sein de l'intergiciel. Son taux de succès est donc de 100%. Cependant, la cohérence finale des messages arrivant à G ou à J subit quand même l'influence des motifs de synchronisation à base de jonctions destinés à la maintenir. Il s'agit précisément des plateaux séparant les composants {C,D}} et {E,F} et {G} dans la figure 6.7b et de ceux entre les composants {E,F} et G ou entre {F,I} et {J} dans 6.8b. Le taux d'erreur introduit par chacun de ces plateaux est identique à celui relevé dans l'exemple de la figure 6.5c.

La figure 6.9 illustre, pour une configuration donnée, l'impact en performance que peuvent avoir les modules les uns sur les autres dans ces schémas. Les multiples plateaux divisent l'application de la figure 6.7b en 4 zones désynchronisées ({A}, {B}, {C,D} et {E,F}) et celle de la figure 6.8b en 6 zones désynchronisées ({A}, {B}, {C,D} et {E,F,I}, {H} et {J}). La figure 6.9 montre que le module le plus lent d'une zone impose sa fréquences aux autres module de la zone.



(a) Spécification avec une contrainte de cohérence non stricte

(b) Une application soumise à une cohérence non stricte

FIGURE 6.7 – Un exemple d'application soumise à une cohérence non stricte.

6.5.2 Application à une simulation de terrain

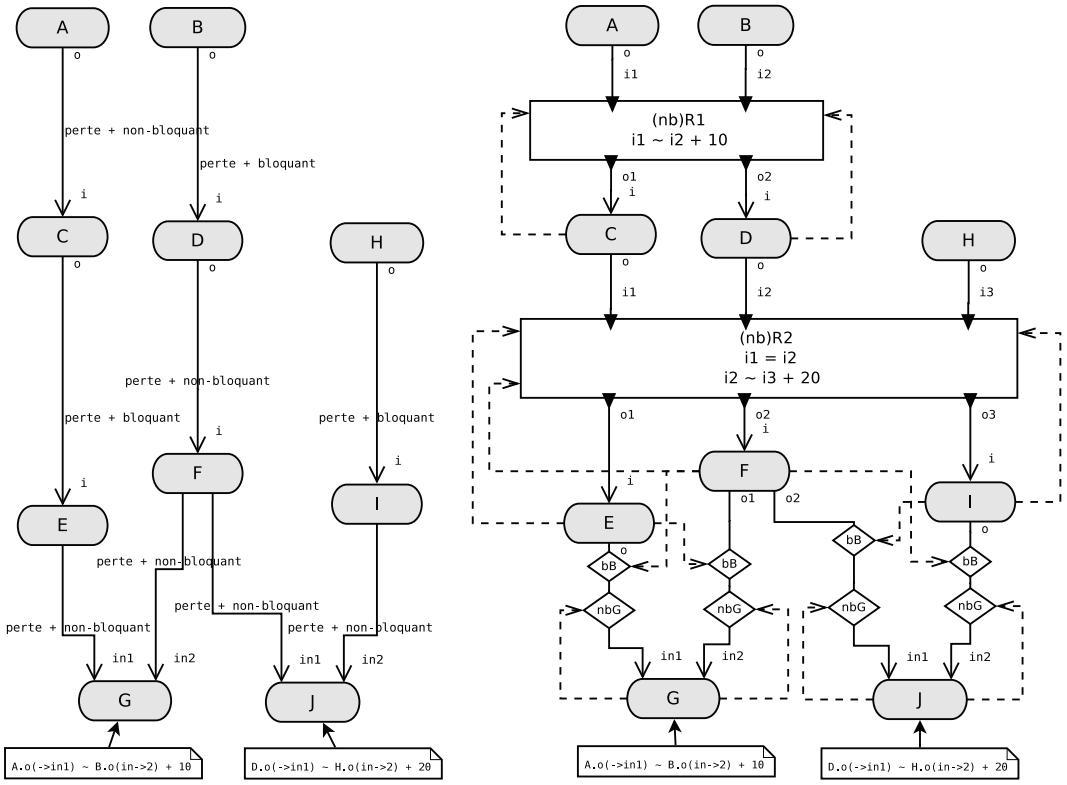
L'annexe E décrit l'application interactive de rendu de terrain GeoCLOD [5]. Réalisée au LIFO et basée sur FlowVR, cette application vise à utiliser le calcul distribué pour permettre la navigation interactive dans un environnement à haut niveau de détail. La figure 6.10 donne un aperçu de l'interface de ce programme.

Construction

Le réseau FlowVR de l'application GeoCLOD, dans sa version élémentaire, est constitué de quatre modules. Tous, hormis Joypad, sont parallélisables.

- HeightFieldBuilder : charge, à partir de données statiques, les informations de relief à plaquer sur les dalles de terrain visibles ;
- TextureBuilder : charge, à partir de données statiques, les textures à plaquer sur les dalles de terrain visibles ;
- Uncompressor : décompresse les données statiques en vue de les afficher ;
- Viewer : Affiche les dalles de terrain par la combinaison des textures et des informations de hauteur décompressées ;
- Joypad : client FlowVR-VRPN [54] permettant à l'utilisateur de se diriger au dessus du terrain grâce à un périphérique d'interaction.

Les modules Uncompressor et Viewer ont été conçus comme deux opérations élémentaires de la phase d'affichage. Pour éviter les artefacts dans cet affichage, il est important



(a) Spécification avec deux contraintes de cohérence non strictes

(b) Une application soumise à deux cohérences non strictes

FIGURE 6.8 – Un exemple d'application soumise à deux cohérences non strictes.

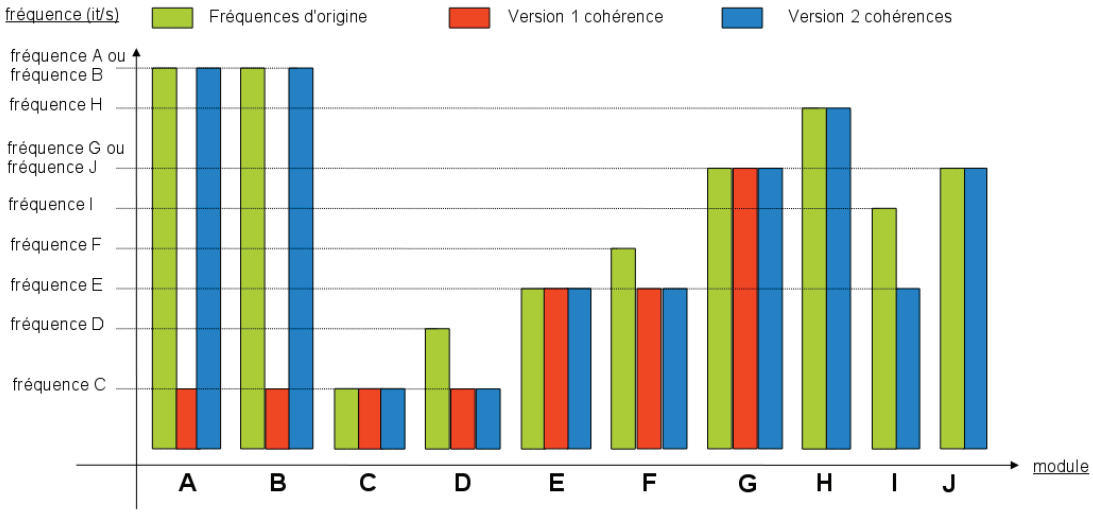


FIGURE 6.9 – Exemples de performances des modules dans des cohérences non strictes.

FIGURE 6.10 – Capture d'écran de l'application GeoCLOD.

que ces deux modules soient parfaitement synchronisés. Nous l'avons donc précisé dans notre fichier de spécification reporté dans l'annexe E en définissant les connexions entre eux comme bloquantes et sans perte. Ensuite, le Viewer communique continuellement aux Builders les informations de frustum [3] et de projection leur indiquant ainsi la position et la direction du point de vue de l'utilisateur. Grâce à ces informations, les Builders déterminent les dalles du terrain à afficher ou affiner. En conséquence, plus les messages portant des données de projection ou de frustum aux Builders sont d'itérations proches, plus court est le temps de rafraîchissement d'une dalle à l'écran. Cette cohérence est indiquée dans le fichier de spécification. Elle est exprimée sous la forme de deux contraintes, l'une concernant les données de frustum et l'autre les données de projection. Afin de s'assurer que cette cohérence se reflète bien sur l'affichage, les ports d'entrée sur lesquels les contraintes sont appliquées appartiennent aussi au composant Viewer :

$$\kappa_1 : Viewer.frustumOutputPort_{\rightarrow Viewer.textureUncompressedDataInputPort} = Viewer.frustumOutputPort_{\rightarrow Viewer.heightFieldUncompressedDataInputPort}$$
$$\kappa_2 : Viewer.projectionOutputPort_{\rightarrow Viewer.textureUncompressedDataInputPort} = Viewer.projectionOutputPort_{\rightarrow Viewer.heightFieldUncompressedDataInputPort}$$

Ces cohérences particulières que l'on peut qualifier de "circulaires" peuvent être traitées par notre méthode.

La figure 6.11 montre le graphe d'application automatiquement généré par notre prototype à partir de la spécification sur l'annexe E. En terme d'utilisation des ressources, cette construction est sensiblement plus optimale que celle utilisée jusqu'ici et reproduite sur la figure 6.12. D'abord, le fait d'utiliser des bGreedy entre Viewer et les Builders assure que ceux-ci ne s'exécuteront qu'à la réception de nouvelles données de point de vue. Pour une raison similaire, les connecteurs entre les Builders et Uncompressor sont également de ce type. En revanche, ceux entre Uncompressor et Viewer sont non bloquants afin que la navigation dans la scène ne soit pas ralentie par des Builders encore occupés.

Les connecteurs utilisés dans la version originale de l'application n'ont pas d'équivalents exacts dans notre modèle. Par exemple, les connecteurs Greedy peuvent desservir plusieurs

3. Région du terrain visible à l'écran.

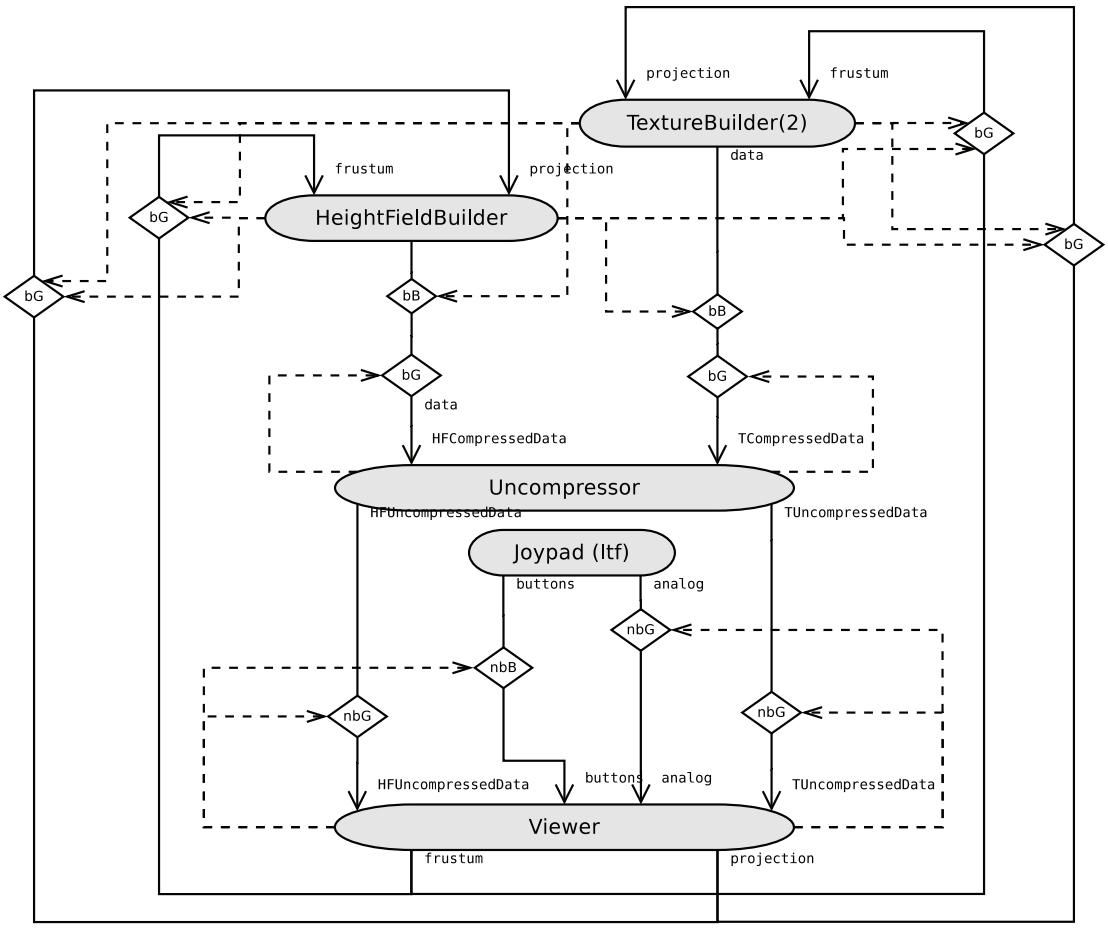


FIGURE 6.11 – Graphe d'application généré pour GeoCLOD.

ports à la fois et, en l'absence de nouveaux messages, ils réémettent le dernier message envoyé et non un message vide. On peut cependant remarquer sur la figure 6.12 les deux blocs désynchronisés que forment, d'une part, les deux Builders et, d'autre part, Uncompressor et Viewer. Contrairement à notre construction, ce découpage permet l'exécution des Builders et d'Uncompressor plusieurs fois sur les mêmes données. De plus, l'absence d'un motif de synchronisation en sortie des deux Builders empêche les deux cohérences strictes κ_1 et κ_2 d'être accomplies.

Performances

Nous avons implémenté via l'API de FlowVR le graphe obtenu et l'avons exécuté sur le cluster Mirev afin de comparer ses performances à celles de l'application originale.

Dans l'application GeoCLOD, la vitesse des deux Builders varie beaucoup au cours de l'exécution et ce, de manière inversement proportionnelle au niveau de détail requis. Ce niveau de détail, lui, augmente avec la proximité entre la caméra et le sol ou les reliefs. La désynchronisation de Viewer et des Builders permet au premier de ne pas pâtir des ralentissements des seconds. Concrètement, dans notre version de l'application, la vitesse minimale observée de Viewer est 2,2 fois supérieure à celle observée dans la version originale. À l'usage, le parcours du terrain est nettement plus fluide pendant les phases de chargement.

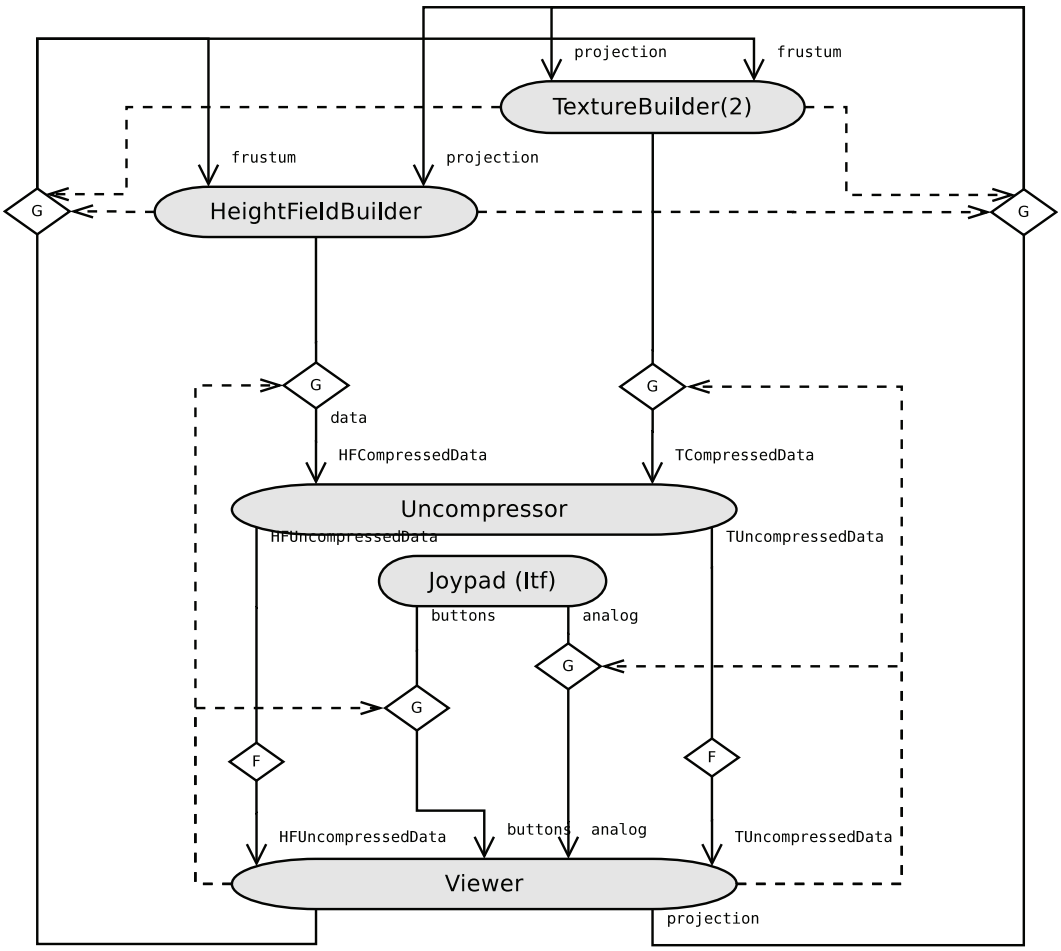


FIGURE 6.12 – Graphe d'application original de GeoCLOD.

6.5.3 Application à une simulation de dynamique moléculaire

L'application de dynamique moléculaire interactive FVnano (contraction de "FlowVR nano") est un projet ANR de laboratoire virtuel basé sur FlowVR. Des membres du LIFO, du CEA/DIF/DSSI, de l'équipe MOAIS de l'INRIA Rhône-Alpes et du Laboratoire de Biochimie Théorique de Paris y participent. L'application FVnano contient une dizaine de modules agencés autour de trois principaux :

GMX C'est le module de simulation. Il est implémenté à partir du moteur de dynamique moléculaire Gromacs [43]. Écrit en langage C, il est transformé en module FlowVR grâce à la librairie FVmoduleAPI. Une définition de molécule doit être chargée au démarrage de ce module. Ensuite, ce dernier émet, à chaque itération, la position actualisée de chaque atome. Un port d'entrée lui permet également de recevoir un vecteur de forces extérieures à appliquer à la molécule.

RenduOpenGL Il s'agit du module de visualisation. Il affiche, dans une fenêtre graphique (voir la figure 6.13) la molécule, l'outil de manipulation (appelé *avatar*) et le vecteur de forces appliquées en temps réel. Cette fenêtre graphique autorise des interactions de type navigation à la souris .

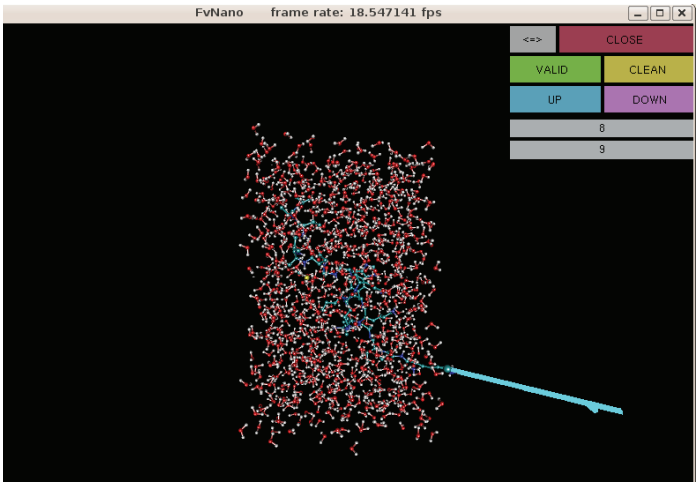


FIGURE 6.13 – Capture d'écran de l'application FVnano.

DeviceManager Ce module gère les interactions utilisateur. Il peut communiquer avec différents périphériques tels qu'un joypad, un bras haptique ou un SpaceNavigator^{TM}grâce à l'interface FlowVR-VRPN [54]. Il envoie aux modules auxquels il est relié la position de l'avatar ainsi que la liste des boutons activés.

Construction

La figure 6.14 montre le graphe d'application automatiquement généré à partir de la spécification reportée dans l'annexe F. Il s'agit d'une version simplifiée de l'application FVnano réelle. N'y apparaissent pas, par exemple, des modules destinés à charger les fichiers d'entrée ou des modules intermédiaires pour la gestion de périphériques d'interaction.

La spécification indique que tous les modules peuvent, dans l'absolu, fonctionner asynchroniquement les uns par rapport aux autres. En même temps, une cohérence stricte y est requise entre l'affichage les atomes sélectionnés et celui de la position de l'avatar :

$$\kappa : DeviceManager.avatar_{\rightarrow RenduOpenGL.avatar} = DeviceManager.avatar_{\rightarrow RenduOpenGL.selection}$$

Le sous-graphe de cette cohérence est illustré par la figure 6.15b. On voit qu'une factorisation des liens de déclenchement a été opérée sur l'application. 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.

Dans le graphe d'application, un cycle synchrone a également été automatiquement débloqué. En effet, suite à l'établissement de la cohérence, un cycle synchrone s'est formé entre les composants AtomSelector, ForceGenerator et GMX. Il est reproduit sur la figure 6.16. L'implémentation de notre méthode de résolution décrite à la section 5.3 a automatiquement sélectionné et rendu non bloquant le connecteur qui n'est impliqué dans aucune cohérence.

Performances

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

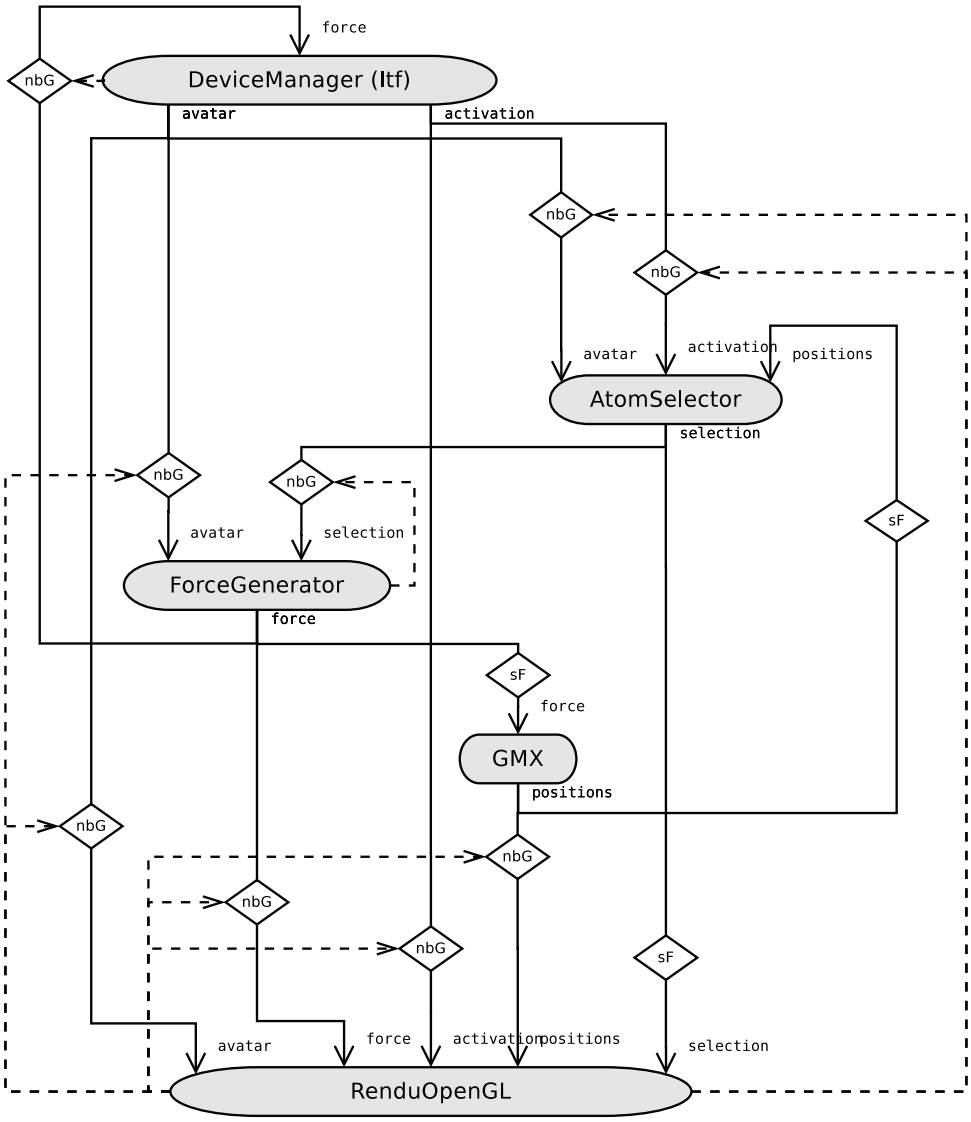
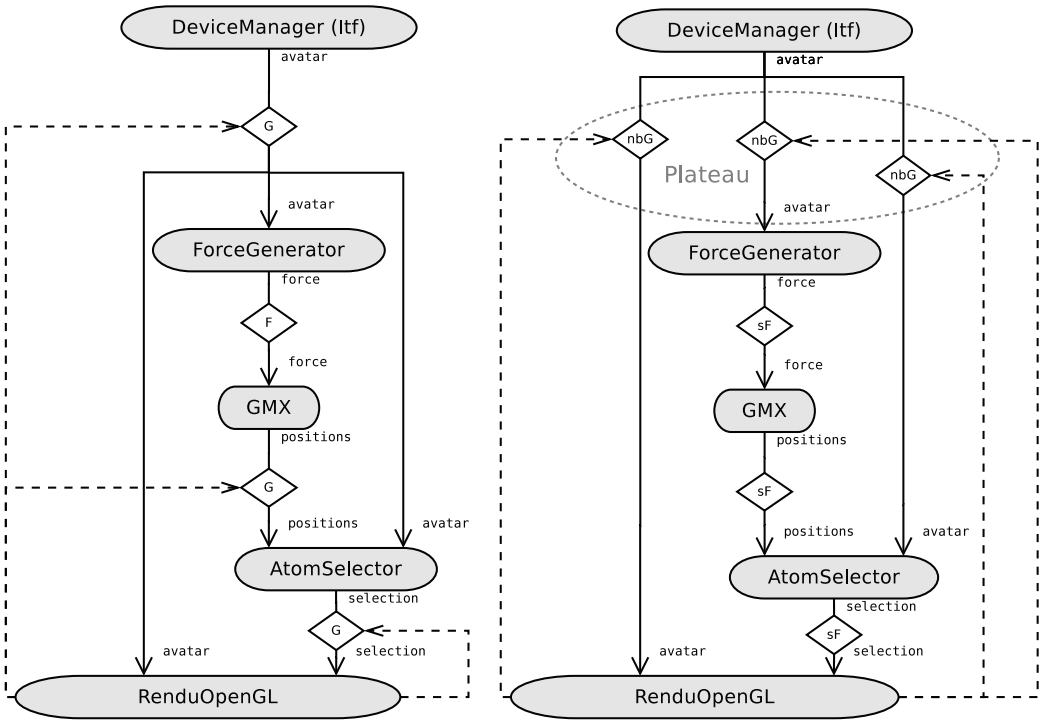


FIGURE 6.14 – Graphe d'application généré pour FVnano.

figure 6.13. 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 6.15a. Cette version utilise les mêmes filtres de FlowVR que la version originale de GeoCLOD vue précédemment. La représentation de la coordination dans ce sous-graphe est cependant fidèle à la réalité.

On observe sur cette construction manuelle que la cohérence stricte, d'après notre définition 10, n'est pas satisfaite. La raison principale est les nombres différents 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, RenduOpenGL peut provoquer la saturation des composants ForceGenerator et AtomSelector s'il est est plus rapide qu'eux. De la même manière, ForceGenerator peut saturer



(a) Sous-graphe à partir de l'application manuellement construite

(b) Sous-graphe à partir de l'application automatiquement construite

FIGURE 6.15 – Sous-graphes pour la contrainte de cohérence dans FVnano.

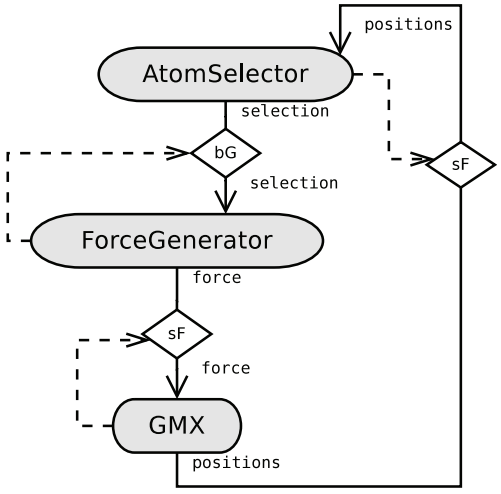


FIGURE 6.16 – Cycle synchrone initialement présent dans FVnano.

GMX. Ce dernier cas s'est d'ailleurs vérifié lors de nos tests et nous avons observé une chute progressive 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.

Dans l'application basée sur notre modèle, la cohérence est obtenue grâce à une SES. Cette cohérence se vérifie à l'usage. Au niveau de la performance, comme le montrent les figures 6.14 et 6.15b, 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 est 91 it/s soit de 10,3% supérieure à celle qu'il a dans la version construite manuellement. Malgré la synchronisation stricte entre ces différents modules, l'application reste parfaitement utilisable et ce, même lorsqu'on freine volontairement la simulation pour pouvoir sélectionner un atome.

6.6 Conclusion

L'expérimentation sur le cluster Mirev présentée à la section 6.5.1 nous a permis de vérifier que nos motifs de coordination restaient fiables dans un contexte distribué. Le faible taux d'erreur que nous imputons aux mécanismes internes de l'intergiciel reste acceptable, en particulier si c'est une cohérence visuelle qui est demandée comme dans les cas réels traités à la section 6.4. Pour les applications où une cohérence stricte de 100% des messages est demandée, nous pouvons alors imaginer remplacer un motif de synchronisation stricte par un régulateur dont la formule de filtrage serait une égalité. Nous discutons cette possibilité plus en détail dans la conclusion de ce mémoire.

Le but de ce chapitre était également d'évaluer l'utilité de la mise en place automatique de nos motifs de coordination et d'estimer leur impact sur la performance de l'application. Nous avons démontré avec l'exemple de GeoCLOD puis celui de FVnano que notre système de construction parvenait à coordonner une application de visualisation scientifique interactive. Ensuite, si nous avons pu vérifier à la section 6.5.2 qu'aucun surcoût n'était introduit par l'utilisation de nos connecteurs, les exemples des sections 6.5.1 et 6.5.3 laissent entrevoir le gain de performance que peut accorder notre modèle lors de la mise en place d'une cohérence.

Ces résultats positifs démontrent l'utilité de notre système dans la coordination d'une application ayant un nombre important de connexions.

Bien qu'elle ne soit pas toujours nécessaire, la cohérence de certains flots de données peut s'imposer pour rendre crédible voire juste utilisable une simulation interactive très désynchronisée. La généricité dans la formulation d'une contrainte (associer un port de sortie à un port d'entrée) permet à l'utilisateur de délimiter les chemins à rendre cohérents. Comme démontré à la section 5.2.2, notre segmentation préserve un maximum de segments désynchronisés dans ces chemins. Cependant, en fonction des vitesses relatives des modules impliqués, il est possible qu'une contrainte de cohérence ralentisse un module essentiel comme la simulation ou qu'un régulateur soit dans l'impossibilité de remplir sa fonction. Idéalement, la connaissance *a priori* des vitesses respectives des modules aiderait, dès la construction, à anticiper certains mauvais fonctionnements. Cet aspect constitue une perspective intéressante de ce travail de thèse. Il est développé, parmi d'autres, dans la conclusion de ce mémoire.

Conclusion

Bilan

Dans ce mémoire, nous avons élaboré un modèle autour d'un composant au comportement naturellement itératif. Ce choix est motivé par le caractère continu dans le temps des applications de visualisation scientifique interactive visées. Nous avons développé ce modèle selon trois axes de recherche correspondant aux trois qualités majeures attendues par ses utilisateurs : abstraction du modèle au moment de la composition et performance et cohérence de l'application. Nous répondons à la première de ces qualités en traduisant tout le modèle en une combinaison de propriétés de haut niveau à l'usage de l'utilisateur. Elle se traduit également dans l'automatisation de certaines opérations d'ordre non-fonctionnel telles que la résolution des interblocages et le dépliage des composants parallèles. La conciliation de la cohérence et de la performance est, elle, abordée sous l'angle de la coordination du graphe d'application. À partir de connecteurs aux sémantiques élémentaires, nous décrivons un ensemble de motifs génériques dont l'assemblage permet de rendre cohérents plusieurs flots de données. Au cœur de ce travail de thèse se trouve le procédé de transformation du graphe d'application en vue de mettre en place ces motifs. La performance de l'application étant, du point de la coordination, tributaire de la désynchronistion des composants, la transformation est orientée pour préserver au mieux cette désynchronisation.

La plupart des systèmes de composition d'applications scientifiques étant destinés à assembler des chaînes de traitements séquentielles, rares sont les travaux ayant porté la coordination de réseaux flot de données au delà du cadre théorique [9, 15, 74]. Nous présentons dans ce mémoire deux applications pratiques de notre approche au domaine de la visualisation scientifique interactive. Bien que le prototype implémentant nos algorithmes de transformation n'aille pas jusqu'à générer les fichiers FlowVR à compiler, les résultats obtenus laissent entrevoir le gain de temps que permet une solution de composition semi-automatique. La performance de l'application automatiquement construire est, elle, au moins équivalente à une version composée manuellement avec l'API de FlowVR.

La génération automatique des fichiers d'application FlowVR fait partie des suites naturelles à apporter à ce travail. Le fichier de spécification devra alors être étendu afin que l'utilisateur puisse y indiquer les emplacements des fichiers-sources de chaque composant ainsi que le répertoire où générer les fichiers d'application. De nombreux autres axes de recherche futurs sont également envisagés pour consolider ou enrichir ce travail.

Un assistant à la composition

À la section 5.2, nous évoquons un certain nombre de prérequis à la validité d'une contrainte de cohérence. Certains concernent la présence et la position d'au moins une connexion autorisant la perte sur chaque chemin d'une cohérence non stricte. Si ces conditions sont vérifiées au début de la transformation, il se peut, parfois, qu'elles ne le soient plus à l'issue de la segmentation, lorsque des contraintes de cohérence strictes et non strictes

sont jointes dans le même sous-graphe. En effet, le processus de transformation que nous décrivons à la section 5.2 est un procédé statique dont l'issue est plus ou moins binaire. En cas de conflit, il contraindra l'utilisateur soit à renoncer à une ou plusieurs cohérences, soit à requalifier en cohérences strictes certaines cohérences non strictes.

Or, dans certains cas, des solutions écartées par la phase de segmentation peuvent s'avérer mieux convenir à l'utilisateur. Il s'agira, le plus souvent, d'un graphe d'application avec un nombre réduit de jonctions ou bien dans lequel le régulateur est placé plus bas sur certains chemins. Ainsi, en faisant varier ces deux paramètres dans le cadre d'une segmentation interactive, l'utilisateur peut aider à résoudre certaines configurations particulières. À défaut, les règles de segmentation fixées à la section 5.2 restent celles résolvant le plus grand nombre de cas automatiquement.

Dans ce contexte, on pourra également envisager une interface graphique pour la composition comme en disposent la grande majorité des systèmes de composition aperçus au chapitre 3.

La réalisation de la cohérence stricte

Comme évoqué à la section 6.5.1, l'implémentation dans FlowVR des motifs de SSS et de SES ne garantit la cohérence stricte que d'environ 99% des couples de messages qui les traversent successivement. Ce taux est amplement acceptable pour une cohérence visuelle telle que demandée dans l'exemple de la section 6.5.3. Dans d'autres circonstances, une cohérence stricte parfaite peut être indispensable au fonctionnement du composant sur lequel elle est définie. Ce dernier peut, par exemple, être un composant de calcul dont la réception d'un seul couple de messages incohérents provoquerait le plantage. Dans un tel cas, nous utiliserions, à la place du couple SSS+SES, un régulateur dont la formule serait de la forme $f : i_1 = i_2$.

Chronologiquement, le régulateur est le dernier objet introduit dans notre modèle [57] et ce, afin de généraliser une définition de la cohérence qui était, au départ, uniquement stricte [56]. Théoriquement, il peut donc se substituer aux mécanismes de SSS et de SES inventés pour réaliser une cohérence stricte à partir de connecteurs élémentaires seulement. Cependant, ces mécanismes gardent un avantage sur le régulateur au regard du déploiement de l'application. En effet, dans de très larges applications distribuées ayant de multiples contraintes de cohérence, un régulateur intervenant dans plusieurs cohérences à la fois constituerait un goulot d'étranglement pour la performance du réseau. À l'opposé, les filtres formant les motifs SSS et SES peuvent être distribués sur des machines différentes.

En fonction de la complexité des futures applications réelles auxquelles nous seront amenés à appliquer ce travail, il convient d'étudier la pertinence de l'usage soit des motifs de synchronisation initiaux soit d'un régulateur pour établir une cohérence ce stricte. Fondamentalement, ce choix sera, de nouveau, celui de la performance ou de la cohérence.

Un modèle de performance

À la section 4.2.2, nous expliquons qu'un connecteur sFIFO ou un connecteur avec perte doit être utilisé si jamais le composant émetteur est plus rapide que le composant récepteur. À défaut, l'utilisation d'un Buffer dans cette situation peut entraîner sa saturation.

Il en va de même pour la définition des contraintes de cohérence. Les motifs de coordination présentés à la section 5.2 ne tiennent pas compte de la fréquence individuelle de chaque composant. Comme évoqué en conclusion du chapitre 6, cette donnée est cependant à considérer par l'utilisateur au moment de définir ses contraintes. Par exemple, si l'écart de fréquences entre deux composants est positif et constant, il est évident qu'à un instant

t, le composant le plus rapide aura produit n fois plus de messages que l'autre. Dans ce contexte, une cohérence stricte entre ces deux composants n'est possible que si le bBuffer en aval du plus lent peut stocker n messages. Concrètement, en se référant au schéma général de la figure 5.5 au chapitre 5, si $freq(A_j) = freq(A_i) + n$ avec $freq(A)$ la fréquence d'un composant A, alors, il faudra que le bBuffer bB_2 de la première SSS ait une capacité d'au moins n messages. Si $freq(A_j) = n \times freq(A_i)$, alors bB_2 devra, dans l'absolu, avoir une capacité illimitée. Une logique similaire est à adopter dans le cas d'une cohérence non stricte. En effet, le régulateur stocke tous les messages entrants jusqu'à ce que ses règles de filtrage soient validées. L'utilisateur doit donc évaluer la pertinence de chaque contrainte qu'il fixe au regard des fréquences respectives des composants sur lesquels il l'applique.

Notre modèle de composants, défini sur une base théorique, s'abstrait totalement de la fréquence de ses composants. Cette fréquence peut, d'ailleurs, varier en fonction du déploiement de l'application. Actuellement, nous admettons que les contraintes de cohérence doivent être définies entre composants aux fréquences proches. Cependant, il sera nécessaire, l'avenir, de faire évoluer notre méthode de composition afin qu'il recoure à un modèle de performances [48] pour prendre ces grandeurs en compte.

Une définition plus générale de la cohérence

Une contrainte de cohérence, telle que l'impose la définition 9, se spécifie entre les chemins allant de deux ports de sortie à un ou deux ports d'entrée distincts d'un même composant. Ces ports d'entrée peuvent, en réalité, être généralisés de telle sorte que la contrainte de cohérence devienne $A.o_{i_{\rightarrow I'}} \circ \alpha \times B.o_{j_{\rightarrow I''}} + \delta$ avec I' et I'' deux ensembles de ports d'entrée dont les éléments de chacun n'appartiennent pas nécessairement au même composant. Notre méthode pour résoudre les contraintes de cohérence s'appliquant sur des chemins de données, elle ne devrait pas beaucoup changer suite à cette extension.

Un autre axe d'évolution pour nos contraintes de cohérence sera de les formuler en fonction d'autres grandeurs que les numéros d'itérations. Les itérations sont un bon indicateur pour comparer les origines temporelles de données provenant d'un même composant. Assimilables aux pas de temps dans un calcul, elles peuvent également servir à comparer les données issues de deux composants similaires. Sur le même modèle, d'autres données numériques transportées par les messages peuvent servir de motifs de régulation : temps, énergie, vélocité, etc. La première problématique à résoudre à ce sujet sera le traitement de valeurs non nécessairement croissantes dans le temps.

Optimiser le déblocage des cycles

La détection et le déblocage automatique des cycles synchrones furent un axe de travail mineur au cours de cette thèse. La solution que nous proposons à la section 5.3 demande le changement définitif d'un ou plusieurs connecteur(s) bloquant(s) en connecteur(s) non bloquant(s). Or, en réalité, il suffit d'injecter un seul message vide initial à l'intérieur d'un cycle synchrone pour le débloquer. Un nouveau type de connecteur, bloquant seulement à partir de sa deuxième itération, pourrait remplir cette fonction. Il conviendra alors d'étudier son intégration à nos motifs de synchronisation au même titre que les autres connecteurs.

Un modèle hiérarchique

Une évolution de notre modèle de composants vers un modèle hiérarchique est envisageable. Afin de respecter le comportement synchrone du composant, un réseau encapsulé dans un composant composite devra lui-même être synchrone. Parmi les points cruciaux

à étudier lors de cette extension seront, alors, la diffusion des signaux de déclenchement de et vers l'intérieur du composant composite ainsi que l'intégration d'un tel composant dans un sous-graphe de cohérence. La structure hiérarchique du modèle de FlowVR [53] facilitera l'implémentation de ce modèle étendu.

Enrichir le couplage

Dans beaucoup de modèles de composants pour application scientifiques [13,37,77], les ports des composants sont marqués des types de données qu'ils véhiculent. Cela constitue un moyen évident de vérifier la validité des connexions établies par l'utilisateur. Un tel enrichissement rendrait notre modèle plus consistant et augmenterait son niveau d'abstraction pour l'utilisateur, surtout si les types de données en question sont exploités à la fois par le langage de spécification et par FVmoduleAPI.

Toujours en terme de couplage, bien que l'annotation constitue notre principale méthode pour transformer un programme en composant, nous avons également exploré [55] une méthode de type passerelle. D'autres expérimentations devront être menées avec FlowVR pour confirmer la viabilité de cette solution.

Quatrième partie

Annexes

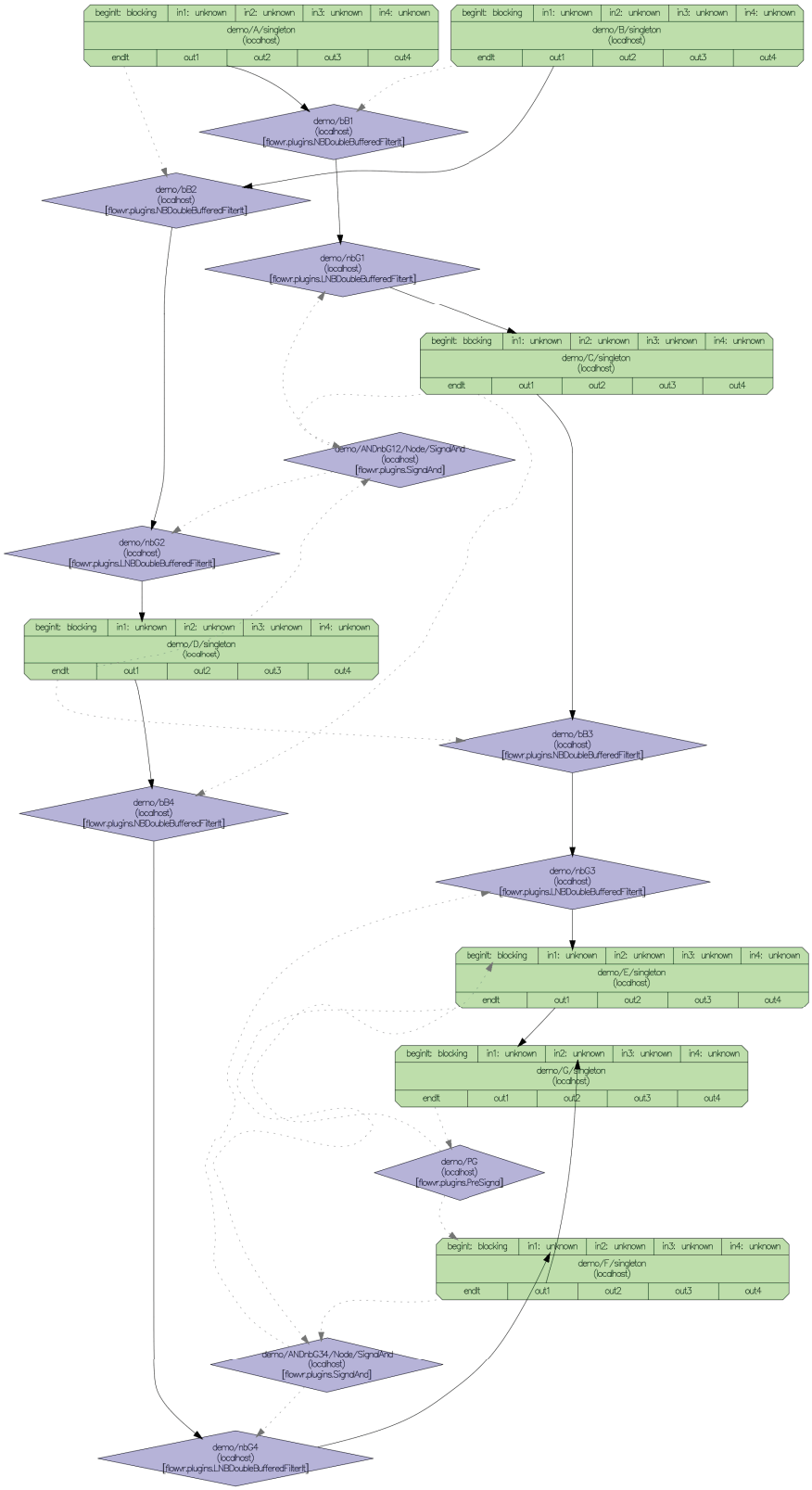
A En-têtes des fonctions de FVmoduleAPI

```
extern "C" {

//////////////////////////
// C/C++ functions
//////////////////////////

/// Add a flowvr input port named "name"
void fv_addinputport(const char* name);

/// Add a flowvr output port named "name"
void fv_addoutputport(const char* name);

/// Initialize the flowvr module and make it parallel if
   necessary
/// Params :
/// nbnodes = the total number of nodes if the module is MPI
   parallel. -1 if the module is FlowVR parallel. 0 disables
   parallelization
/// rank = the current node's rank number if the module MPI
   parallel
int fv_initmodule(int nbnodes, int rank);

/// If the module is FlowVR parallel, returns the rank of the
   current process
int fv_getnoderank();

/// If the module is FlowVR parallel, returns the total number
    of processes
int fv_getprocnumber();

/// Returns whether the module is currently running or not
/// Returns the result of module->wait()
int fv_moduleruns();

/// Terminate the flowvr module
void fv_closemodule();

/// Put in variable "data" the data received on inputport "
   port"
/// Enough memory space must be allocated to "data"
/// Params :
/// data = pointer to the module's concerned variable
/// port = name of the module's inputport
int fv_get(void* data, const char* port);

/// Only gets the size in octets of the data received on port
   "port"
```

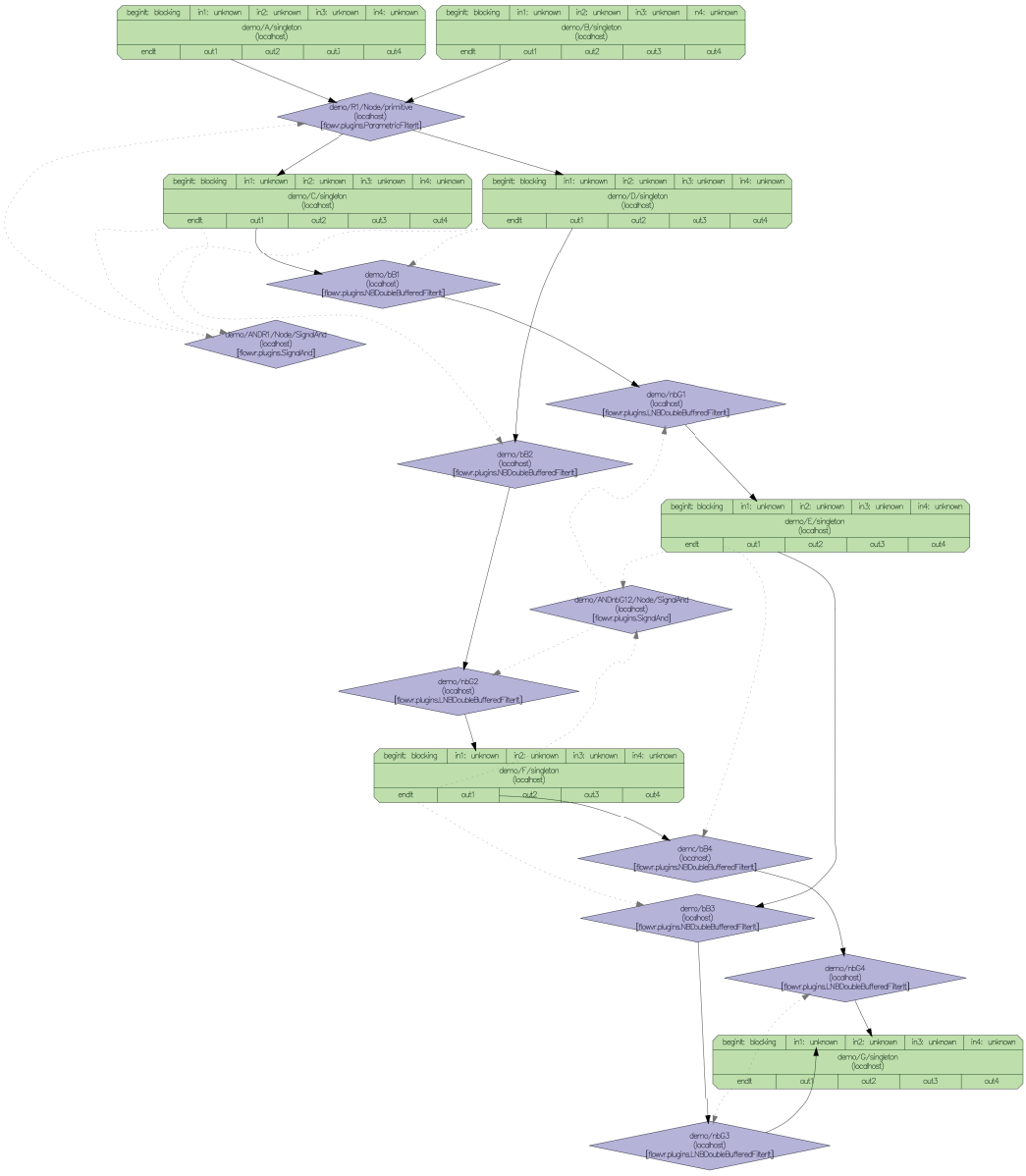
```
int fv_getdatasize(const char* port);

/// Send on outputport "port" the content of variable "data"
   of size "size"
/// Params :
/// data = pointer to the module's concerned variable
/// size = size in octets of the module's concerned variable
/// port = name of the module's inputport
void fv_put(const void* data, int size, const char* port);

/// Add a stamp information of type int named "name" and
   containing "size" elements to port "port"
/// Params :
/// name = name given to the stamp information
/// size = number of elements of the stamp
/// port = name of the port to add the stamp to
void fv_addintstamp(char* name, int size, const char* port);

/// Add a stamp information of type float named "name" and
   containing "size" elements to port "port"
void fv_addfloatstamp(char* name, int size, const char* port);

/// Add a stamp information of type string named "name" and
   containing "size" elements to port "port"
void fv_addstringstamp(char* name, int size, const char* port)
   ;

/// Set the element of index "index" of stamp "stamp" of port
   "port" to the value of "data"
/// Params :
/// data = pointer to the module's variable from which the
   stamp information should be copied
/// port = name of the port the stamp is related to
/// stamp = name of the stamp information
/// index = index of the stamp element the data should be
   copied to. If there is only one element in the stamp, index
   should be 0
int fv_writestamp(const void* data, const char* port, const
   char* stamp, int index);

// Set the module's variable "data" to the value of the
   element of index "index" of stamp "stamp" of port "port"
// Params :
// data = pointer to the module's variable to which the stamp
   information should be copied
// port = name of the port the stamp is related to
// stamp = name of the stamp information
// index = index of the stamp element the data should be
   copied from. If there is only one element in the stamp,
```

```
   index should be 0
int fv_readstamp(void* data, const char* port, const char*
   stamp, int index);

//////////////////////
// Fortran functions
//////////////////////

void fv_addoutputport_(char* name, int length);

void fv_addinputport_(char* name, int length);

void fv_initmodule_(int* nbnodes, int* rank);

int fv_moduleruns_();

void fv_closemodule_();

int fv_get_(void* data, const char* port, int length);

int fv_getdatasize_(const char* port, int length);

void fv_put_(const void* data, int* size, const char* port,
   int length);

void fv_addintstamp_(char* name, int* size, const char* port,
   int nlength, int plength);

void fv_addfloatstamp_(char* name, int* size, const char* port
   , int nlength, int plength);

void fv_addstringstamp_(char* name, int* size, const char*
   port, int nlength, int plength);

int fv_writestamp_(const void* data, const char* port, const
   char* stamp, int* index, int plength, int slength);

int fv_readstamp_(void* data, const char* port, const char*
   stamp, int* index, int plength, int slength);

}
```

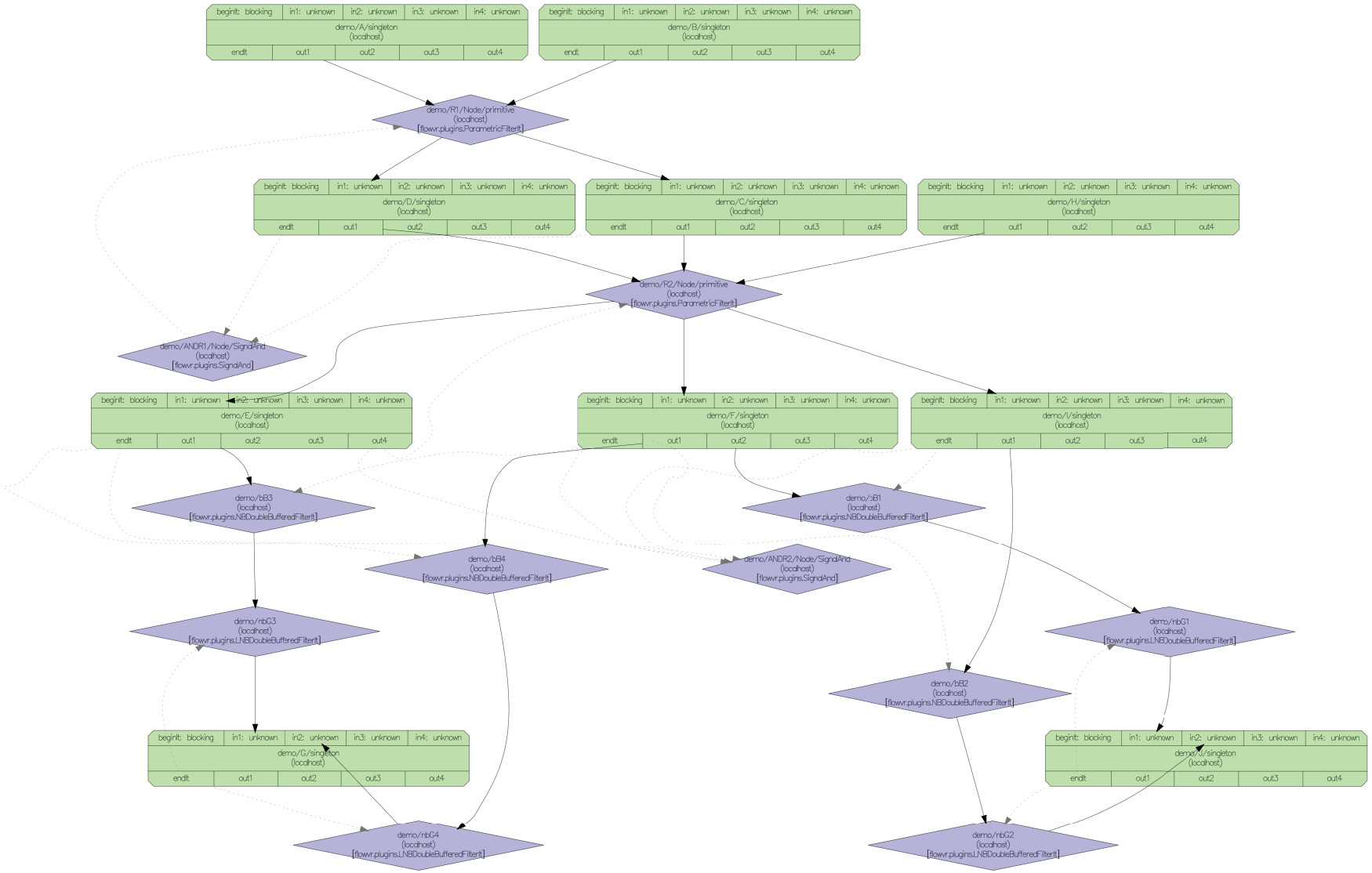
B Réseau FlowVR d'une application générique soumise à une cohérence stricte



C Réseau FlowVR d'une application générique soumise à une cohérence non stricte



D Réseau FlowVR d'une application générique soumise à deux cohérences non strictes



E Fichier de spécification de l'application GeoCLOD

```
<components>
        <component name="joypad" isinteractive="yes" nbinstances=1>
        <ports>
                <outputport name="buttons"/>
            <outputport name="analog"/>
        </ports>
    </component>
    <component name="heightFieldBuilder" isinteractive="no" nbinstances
        =2>
        <ports>
            <inputport name="projection"/>
            <inputport name="frustum"/>
            <outputport name="data"/>
        </ports>
    </component>
    <component name="textureBuilder" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="projection"/>
            <inputport name="frustum"/>
            <outputport name="data"/>
        </ports>
    </component>
    <component name="uncompressor" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="heightFieldCompressedDataInputPort"/>
            <inputport name="textureCompressedDataInputPort"/>
            <outputport name="heightFieldUncompressedDataOutputPort"/>
            <outputport name="textureUncompressedDataOutputPort"/>
         </ports>
    </component>
    <component name="viewer" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="heightFieldUncompressedDataInputPort"/>
            <inputport name="textureUncompressedDataInputPort"/>
            <inputport name="JoypadButton"/>
            <inputport name="JoypadAnalog"/>
            <outputport name="projectionOutputPort"/>
            <outputport name="frustumOutputPort"/>
        </ports>
            <coherences>
                        <coherence operator="=" factor=1 offset=0>
                        <inputport name="
                            heightFieldUncompressedDataInputPort"
                            component="viewer" outputport="
                            frustumOutputPort" />
                        <inputport name="
                            textureUncompressedDataInputPort" component
                            ="viewer" outputport="frustumOutputPort" />
                </coherence>
                <coherence operator="=" factor=1 offset=0>
                        <inputport name="
                            heightFieldUncompressedDataInputPort"
                            component="viewer" outputport="
```

```
                        projectionOutputPort" />
                    <inputport name="
                        textureUncompressedDataInputPort" component
                        ="viewer" outputport="projectionOutputPort"
                         />
                </coherence>
            </coherences>
    </component>
</components>
<connections>
<connection sender="joypad" outputport="buttons" receiver="viewer"
    inputport="JoypadButton" blocking="no" allowmsgloss="no" />
<connection sender="joypad" outputport="analog" receiver="viewer"
    inputport="JoypadAnalog" blocking="no" allowmsgloss="yes" />
<connection sender="heightFieldBuilder" outputport="data" receiver="
    uncompressor" inputport="heightFieldCompressedDataInputPort"
    blocking="yes" allowmsgloss="yes" />
<connection sender="textureBuilder" outputport="data" receiver="
    uncompressor" inputport="textureCompressedDataInputPort" blocking="
    yes" allowmsgloss="yes" />
<connection sender="uncompressor" outputport="
    heightFieldUncompressedDataOutputPort" receiver="viewer" inputport=
    "heightFieldUncompressedDataInputPort" blocking="no" allowmsgloss="
    yes" />
<connection sender="uncompressor" outputport="
    textureUncompressedDataOutputPort" receiver="viewer" inputport="
    textureUncompressedDataInputPort" blocking="no" allowmsgloss="yes"
    />
<connection sender="viewer" outputport="projectionOutputPort" receiver=
    "heightFieldBuilder" inputport="projection" blocking="yes"
    allowmsgloss="yes" />
<connection sender="viewer" outputport="projectionOutputPort" receiver=
    "textureBuilder" inputport="projection" blocking="yes" allowmsgloss
    ="yes" />
<connection sender="viewer" outputport="frustumOutputPort" receiver="
    heightFieldBuilder" inputport="frustum" blocking="yes" allowmsgloss
    ="yes" />
<connection sender="viewer" outputport="frustumOutputPort" receiver="
    textureBuilder" inputport="frustum" blocking="yes" allowmsgloss="
    yes" />
</connections>
```

F Fichier de spécification de l'application FVNano

```
<components>
    <component name="deviceManager" isinteractive="yes" nbinstances=1>
        <ports>
                <inputport name="force"/>
            <outputport name="avatar"/>
            <outputport name="activation"/>
        </ports>
    </component>
    <component name="GMX" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="force"/>
            <outputport name="positions"/>
         </ports>
    </component>
    <component name="atomselector" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="positions"/>
            <inputport name="avatar"/>
            <inputport name="activation"/>
            <outputport name="selection"/>
         </ports>
    </component>
   <component name="forcegenerator" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="selection"/>
            <inputport name="avatar"/>
            <inputport name="activation"/>
            <outputport name="force"/>
         </ports>
    </component>
    <component name="renduopengl" isinteractive="no" nbinstances=1>
        <ports>
            <inputport name="positions"/>
            <inputport name="selection"/>
            <inputport name="force"/>
            <inputport name="avatar"/>
            <inputport name="activation"/>
         </ports>
                 <coherences>
                <coherence operator="=" factor=1 offset=0>
                <inputport name="Avatar" component="deviceManager"
                    outputport="AvatarOut" />
                <inputport name="selection" component="deviceManager"
                    outputport="AvatarOut" />
                </coherence>
          </coherences>
    </component>
</components>
<connections>
<connection sender="deviceManager" outputport="activation" receiver="
    atomselector" inputport="activation" blocking="no" allowmsgloss="
    yes" />
```

```
<connection sender="deviceManager" outputport="activation" receiver="
    renduopengl" inputport="activation" blocking="no" allowmsgloss="yes
    " />
<connection sender="deviceManager" outputport="avatar" receiver="
    atomselector" inputport="avatar" blocking="no" allowmsgloss="yes" /
    >
<connection sender="deviceManager" outputport="avatar" receiver="
    forcegenerator" inputport="avatar" blocking="no" allowmsgloss="yes"
     />
<connection sender="deviceManager" outputport="avatar" receiver="
    renduopengl" inputport="avatar" blocking="no" allowmsgloss="yes" />
<connection sender="gmx" outputport="positions" receiver="atomselector"
     inputport="positions" blocking="yes" allowmsgloss="yes" />
<connection sender="gmx" outputport="positions" receiver="renduopengl"
    inputport="positions" blocking="no" allowmsgloss="yes" />
<connection sender="forcegenerator" outputport="force" receiver="gmx"
    inputport="force" blocking="yes" allowmsgloss="yes" />
<connection sender="forcegenerator" outputport="force" receiver="
    renduopengl" inputport="force" blocking="no" allowmsgloss="yes" />
<connection sender="forcegenerator" outputport="force" receiver="
    deviceManager" inputport="force" blocking="no" allowmsgloss="yes" /
    >
<connection sender="atomselector" outputport="selection" receiver="
    forcegenerator" inputport="selection" blocking="yes" allowmsgloss="
    yes" />
<connection sender="atomselector" outputport="selection" receiver="
    renduopengl" inputport="selection" blocking="no" allowmsgloss="yes"
     />
</connections>
```

Bibliographie

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

[2] J. Allard and B. Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference (VR 2006)*, pages 89–96. IEEE, 2006.

[3] Jérémie Allard, Jean-Denis Lesage, and Bruno Raffin. Modularity for Large Virtual Reality Applications. *Presence : Teleoperators and Virtual Environments*, 19(2) :142–161, April 2010.

[4] Farhad Arbab, Tom Chothia, Sun Meng, and Y.J. Moon. Component connectors with QoS guarantees. In *Coordination Models and Languages*, pages 286–304. Springer, 2007.

[5] Simon Arvaux, Joeffrey Legaux, Sébastien Limet, Emmanuel Melin, and Sophie Robert. Parallel lod for static and dynamic generic geo-referenced data. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, VRST '08, pages 301–302, New York, NY, USA, 2008. ACM.

[6] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. Available at <http ://lpsolve.sourceforge.net/5.5/>.

[7] D E Bernholdt, G. Kumfert, T G W Epperly, J A Kohl, L C McInnes, S. Parker, and J. Ray. How the common component architecture advances computational science. *Journal of Physics : Conference Series*, 46(1) :479–493, September 2006.

[8] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, and Others. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2) :163, May 2006.

[9] B. Biornstad, Cesare Pautasso, and Gustavo Alonso. Control the flow : How to safely compose streaming services into business processes. In *IEEE International Conference on Services Computing, 2006. SCC'06*, pages 206–213, 2006.

[10] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1) :39–59, February 1984.

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

[12] Matthieu Chavent, Antoine Vanel, Alex Tek, Bruno Levy, Sophie Robert, Bruno Raffin, and Marc Baaden. GPU-accelerated atom and dynamic bond visualization using hyperballs : A unified algorithm for balls, sticks, and hyperboloids. *Journal of computational chemistry*, pages 1–12, July 2011.

[13] Eran Chinthaka, Jaliya Ekanayake, David Leake, and Beth Plale. CBR Based Workflow Composition Assistant. In *2009 Congress on Services - I*, pages 352–355. IEEE, July 2009.

[14] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation : Practice and Experience*, 18(10) :1021–1037, August 2006.

[15] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76(8) :681–710, August 2011.

[16] L. Clarke, I. Glendinning, and R. Hempel. The MPI Message Passing Interface Standard. In *Programming environments for massively parallel distributed systems : working conference of the IFIP*, page 213. Birkhäuser, 1994.

[17] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, 2009.

[18] Brad J Cox. *Object oriented programming : an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[19] Daniel Crawl and I. Altintas. A provenance-based fault tolerance mechanism for scientific workflows. *Provenance and Annotation of Data and Processes*, pages 152–159, 2008.

[20] Silvia Crivelli, Oliver Kreylos, Bernd Hamann, Nelson Max, and Wes Bethel. ProteinShop : a tool for interactive protein manipulation and steering. *Journal of computer-aided molecular design*, 18(4) :271–85, April 2004.

[21] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1) :11 – 24, 1988.

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

[23] T.A. DeFanti, M.D. Brown, and B.H. McCormick. Visualization : Expanding scientific and engineering research opportunities. *Computer*, 22(8) :12–16, 1989.

[24] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.

[25] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[26] Wolfgang Emmerich and Nima Kaveh. 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*, pages 691–692, New York, NY, USA, 2002. ACM Press.

[27] Robert Englander. *Developing Java beans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[28] A. Esnard, M. Dussere, and Olivier Coulaud. A time-coherent model for the steering of parallel simulations. In *Euro-Par 2004 Parallel Processing*, pages 90–97. Springer, 2004.

[29] Aurelien Esnard, Nicolas Richart, and Olivier Coulaud. A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations. *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 7–14, October 2006.

[30] Jean Favre and Mario Valle. AVS and AVS/Express. In Chuck Hansen and Chris Johnson, editors, *The Visualization Handbook*, pages 655–672. Academic Press, December 2004.

[31] W Fisher and J Gilbert. Filenet : A distributed system supporting workflo ; a flexible office procedures control language. In *IEEE Computer Society Office Automation Symposium*, pages 247–249, 1987.

[32] T. Forkert, G.K. Kloss, C. Krause, and a. Schreiber. Techniques for Wrapping Scientific Applications to CORBA Components. *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, pages 100–108, 2004.

[33] Philippe Fuchs. *Le Traité de la Réalité Virtuelle.* Presses de l'Ecole des Mines, March 2006.

[34] Mohamed Gad-el Hak and Matthias H. Buschmann. Turbulent boundary layers : is the wall falling or merely wobbling ? *Acta Mechanica*, 218(3-4) :309–318, December 2010.

[35] ER Gansner. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11) :1203–1233, September 2000.

[36] J.W. Gibbs. *A method of geometrical representation of the thermodynamic properties of substances by means of surfaces.* Connecticut Academy of Arts and Sciences, 1873.

[37] Y Gil, V Ratnakar, and C. Fritz. Assisting scientists with complex data analysis tasks through semantic workflows. In *Proceedings of the AAAI Fall Symposium on Proactive Assistant Agents, Arlington, VA*, 2010.

[38] Yolanda Gil. Workflow composition : Semantic representations for flexible automation. *Workflows for e-Science*, pages 244–257, 2007.

[39] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit : Design and applications. In *Vector and Parallel Processing-VECPAR*, pages 1–31. Citeseer, 2002.

[40] T Goodale and I Taylor. Integrating Cactus simulations within Triana workflows. *Proceedings of 13th Annual Mardi Gras*, pages 47–53, 2005.

[41] B Guillerminet, M Airaj, P Huynh, and G Huysmans. Integrated tokamak modelling : Infrastructure and Software Integration Project. *Fusion Engineering and Design*, 83 :442–447, 2008.

[42] A. Henderson and J. Ahrens. *The ParaView guide.* Kitware, 2004.

[43] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4 : Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation*, 4(3) :435–447, March 2008.

[44] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, January 1995.

[45] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide.* Kitware, Inc. ISBN 1-930934-15-7, http ://www.itk.org/ItkSoftwareGuide.pdf, second edition, 2005.

[46] D.J. Jablonowski, J.D. Bruner, B Bliss, and R.B. Haber. VASE : The visualization and application steering environment. In *Supercomputing'93. Proceedings*, pages 560–569, New York, New York, USA, 1993. IEEE.

[47] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[48] Sylvain Jubertie. *Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées*. PhD thesis, Université d'Orléans, December 2007.

[49] G. Kloss and Andreas Schreiber. Provenance implementation in a scientific simulation environment. *Provenance and Annotation of Data*, pages 37–45, 2006.

[50] James Arthur Kohl, Torsten Wilde, and David E. Bernholdt. Cumulvs : Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. *IJHPCA*, 20(2) :255–285, 2006.

[51] O Kreylos, AM Tesdall, B Hamann, JK Hunter, and KI Joy. Interactive visualization and steering of CFD simulations. In *Proceedings of the symposium on Data Visualisation 2002*, pages 25–34. Eurographics Association, 2002.

[52] K.K. Lau, P. Velasco Elizondo, and Zheng Wang. Exogenous connectors for software components. *Component-Based Software Engineering*, pages 221–245, 2005.

[53] Jean-Denis Lesage and Bruno Raffin. A hierarchical component model for large parallel interactive applications. *The Journal of Supercomputing*, July 2008.

[54] S. Limet and S. Robert. FlowVR-VRPN : first experiments of a VRPN/FlowVR coupling. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, pages 251–252. ACM New York, NY, USA, 2008.

[55] S. Limet, S. Robert, and A. Turki. FlowVR-SciViz : a component-based framework for interactive scientific visualization. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, pages 1–9. ACM, 2009.

[56] S. Limet, Sophie Robert, and Ahmed Turki. Coherence and Performance for Interactive Scientific Visualization Applications. In *Software Composition*, pages 149–164. Springer, 2011.

[57] S. Limet, Sophie Robert, and Ahmed Turki. Controlling an iteration-wise coherence in dataflow. In *8th International Symposium on Formal Aspects of Component Software*. Springer, To appear.

[58] Cui Lin, Shiyong Lu, Xubo Fei, Artem Chebotko, Darshan Pai, Zhaoqiang Lai, Farshad Fotouhi, and Jing Hua. A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution. *IEEE Transactions on Services Computing*, 2(1) :79–92, January 2009.

[59] Books LLC. *Fortran Compilers : Gnu Compiler Collection, Ibm Visualage, Intel Fortran Compiler, Low Level Virtual Machine, Open64, Pathscale, Gfortran, F2c*. Books Nippan, 2010.

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

[61] J. L. Lumley. Some comments on turbulence. *Physics of Fluids A : Fluid Dynamics*, 4(2) :203, 1992.

[62] Robert Marshall, Jill Kempf, Scott Dyer, and Chieh-Cheng Yen. Visualization methods and simulation steering for a 3D turbulence model of Lake Erie. *Proceedings of the 1990 symposium on Interactive 3D graphics - SI3D '90*, pages 89–97, 1990.

[63] K. Martin and B. Hoffman. Mastering cmake : A cross-platform build system, second edition. 01 2004.

[64] James Clerk Maxwell and P. M. Harman. *The scientific letters and papers of James Clerk Maxwell / edited by P.M. Harman.* Cambridge University Press, Cambridge [England] ; New York :, 1990.

[65] Anthony Mayer, S. McGough, Nathalie Furmento, William Lee, Steven Newhouse, and John Darlington. ICENI dataflow and workflow : Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, volume 634, pages 627–634. Citeseer, 2003.

[66] Rohan J. McAdam. Continuous interactive simulation : Engaging the human sensorymotor system in understanding dynamical systems. *Procedia Computer Science*, 1(1) :1691–1698, May 2010.

[67] B. H. McCormick. Visualization in scientific computing. *SIGBIO Newsl.*, 10 :15–21, March 1988.

[68] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[69] Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1) :119–129, February 1999.

[70] Eduardo Ogasawara, Daniel de Oliveira, Fernando Chirigati, Carlos Eduardo Barbosa, Renato Elias, Vanessa Braganholo, Alvaro Coutinho, and Marta Mattoso. Exploring many task computing in scientific workflows. *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers - MTAGS '09*, pages 1–10, 2009.

[71] Eduardo Ogasawara, Carlos Paulino, Leonardo Murta, C. Werner, and M. Mattoso. Experiment line : software reuse in scientific workflows. In *Scientific and Statistical Database Management*, pages 264–272. Springer, 2009.

[72] OMG. Omg unified modeling language (omg uml) infrastructure version 2.3. Technical Report formal/2010-05-03, 2010.

[73] S.G. Parker, K. Damevski, A. Khan, A. Swaminathan, and C.R. Johnson. The SCIJump Framework for Parallel and Distributed Scientific Computing. *Advanced Computational Infrastructures for Parallel/Distributed Adaptive Applications*, 2007.

[74] Cesare Pautasso and Gustavo Alonso. Parallel computing patterns for Grid workflows. In *2006 Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE, June 2006.

[75] J. Petri. *Netbeans Platform 6.9 Developer's Guide.* Packt Publishing, Limited, 2010.

[76] Clifford A. Pickover and Stuart K. Tewksbury. *Frontiers of scientific visualization.* Wiley, 1994.

[77] Jun Qin, Thomas Fahringer, and Radu Prodan. A novel graph based approach for automatic composition of high quality grid workflows. *Proceedings of the 18th ACM international symposium on High performance distributed computing - HPDC '09*, page 167, 2009.

[78] S. Rathmayer and M. Lenke. A tool for on-line visualization and interactive steering of parallel hpc applications. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 181–186. IEEE, 1997.

[79] O. Reynolds. On the Dynamical Theory of Incompressible Viscous Fluids and the Determination of the Criterion. *Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences*, 186 :123–164, January 1895.

[80] Nicolas Richart, Aurelien Esnard, and Olivier Coulaud. Toward a Computational Steering Environment for Legacy Coupled Simulations. *Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07)*, pages 43–43, July 2007.

[81] G. Scherp and W. Hasselbring. Towards a model-driven transformation framework for scientific workflows. *Procedia Computer Science*, 1(1) :1519–1526, May 2010.

[82] W. Richard Stevens. *UNIX network programming, volume 2 (2nd ed.) : interprocess communications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[83] F.W. Taylor. *The principles of Scientific Management*. Harper and Row, New York, 1911.

[84] I. Taylor, M. Shields, and I. Wang. Distributed p2p computing within triana : A galaxy visualization test case. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[85] J. Vanegue, T. Garnier, J. Auto, S. Roy, and R. Lesniak. Next generation debuggers for reverse engineering. *Black Hat Europe, Amsterdam, Netherlands*, pages 1–29, 2007.

[86] J. Vetter and K. Schwan. Techniques for high-performance computational steering. *IEEE Concurrency*, 7(4) :63–74, 1999.

[87] Ian Wang, Ian Taylor, Tom Goodale, Andrew Harrison, and Matthew Shields. gridMonSteer : Generic architecture for monitoring and steering legacy applications in grid environments. In *Proceedings of the UK e-Science All Hands Meeting*, 2006.

[88] D.M. Weinstein, Steven Parker, Jenny Simpson, Kurt Zimmerman, and G. Jones. Visualization in the scirun problem-solving environment. *The Visualization Handbook*, pages 615–632, 2005.

[89] Petra Wenisch, Christoph Van Treeck, André Borrmann, Ernst Rank, and Oliver Wenisch. Computational steering on distributed systems : Indoor comfort simulations as a case study of interactive CFD on supercomputers. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4) :275–291, August 2007.

[90] A Wierse. Collaborative visualization based on distributed data objects. *Database Issues for Data Visualization*, pages 208–219, 1996.

[91] Ustun Yildiz, Adnene Guabtni, and A.H.H. Ngu. Towards scientific workflow patterns. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, New York, USA, 2009. ACM.

[92] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4) :171–200, January 2006.

[93] Reza Ziaei and Gul Agha. SynchNet : a petri net based coordination language for distributed objects. In *Generative Programming and Component Engineering*, pages 324–343. Springer, 2003.

Ahmed TURKI

Un modèle pour la composition d'applications de visualisation et d'interaction continue avec des simulations scientifiques

Résumé : La simulation informatique est un outil incontournable dans les sciences expérimentales. La puissance de calcul croissante des ordinateurs associée au parallélisme et aux avancées dans la modélisation mathématique des phénomènes physiques permet de réaliser virtuellement des expériences de plus en plus complexes. De plus, l'émergence de la programmation GPU a considérablement accru la qualité et la rapidité de l'affichage. Ceci a permis de démocratiser la visualisation sous forme graphique des résultats de simulation. La visualisation scientifique peut être passive : l'utilisateur peut suivre l'évolution de la simulation ou bien observer ses résultats après que le calcul soit terminé. Elle peut aussi être interactive lorsque le chercheur peut agir sur la simulation alors qu'elle se déroule. Créer de telles applications complexes n'est cependant pas à la portée de tout scientifique non informaticien. La programmation par composants est, depuis des années, mise en avant comme une solution à ce problème. Elle consiste à construire des applications en interconnectant des programmes exécutant des tâches élémentaires. Ce mémoire présente un modèle de composants et une méthode de composition d'applications de visualisation scientifique interactive. Elle s'intéresse, en particulier, à la conciliation de deux contraintes majeures dans la coordination de ces applications : la performance et la cohérence.

Mots clés : visualisation scientifique, programmation par composants, coordination, provenance.

A model for composing applications of visualization and continuous interaction with scientific simulations

Abstract : Computer simulation is an essential tool in experimental sciences. The increasing computing power, parallelism and the advances in the mathematical modeling of physical phenomena allow to virtually run always more complex experiments. In addition, the rise of GPU programming has greatly increased the quality and performance of display. This has allowed to spread the graphical visualization of simulation results. Scientific visualization can be passive : the user can only follow the simulation's progress or observe its results when it is done. It can also be interactive in which case the researcher can act on the simulation while it is running. Creating such complex applications can, however, be tedious for non-computer-scientists. Component-based development is, for years, highlighted as a solution to this problem. It consists in building applications by interconnecting small programs completing elementary tasks. This thesis presents a component model and a method for composing interactive scientific visualization applications. It particularly focuses on the balance between two major constraints of these applications : performance and coherence.

Keywords : scientific visualization, component-based development, coordination, provenance.

LIFO - Bâtiment IIIA Rue Léonard de Vinci B.P.
6759 F-45067 ORLÉANS Cedex 2