4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Coherence and Performance for Interactive Scientific Visualization Applications

Sébastien Limet, Sophie Robert, Ahmed Turki
LIFO, Université d'Orléans

Rapport n^o **RR-2011-02**

Coherence and Performance for Interactive Scientific Visualization Applications

Sébastien Limet, Sophie Robert, and Ahmed Turki

Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, France *

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

Keywords: Composition, Coherence, Coordination, Synchronization

1 Introduction

The interactive visualization of simulations helps scientists better understand the phenomena they study. In [5] for example, biochemists describe how it can unveil some interactions between complex molecular assemblies. The observer can then become an actor by applying forces to the atoms. The intended interactivity in these applications is not limited to a passive manipulation of the graphical output. It is rather active and its effects are propagated throughout the whole running application. As an example, one can think of pushing and pulling atoms during a molecular simulation.

The development of interactive scientific visualizations is however hampered by the complexity of the software developments it needs. Scientific simulation specialists are seldom experts in 2D/3D visualization or in control device programming. Component-based development promotes collaborative work in this area since it allows each specialist to independently develop the parts he is skilled in. The component approach has been widely studied for scientific computing. These approaches generally follow a workflow model. A variety of Scientific Workflow Management Systems (SWMSs) [17] are proposed to design,

* This work is supported by the french ANR project FvNano.

generate, deploy and execute scientific applications which consists in carrying out an overall process over a dataset through a sequence of finite steps [4, 7, 8]. Some of them are extended to provide control constructs that allow branching, looping and parallelism. These control constructs are based on control nodes in [15], on control links in [11], on directors in [12] or on adaptation connectors in [6]. But in all cases the user has the task of manually specifying the control behaviour of the components and of the overall application by instanciating these control constructs.

Moreover, the lack of explicit support of data streams in SWMSs hampered application development in some scientific domains. Now, initiatives are being taken, either by extending a current workflow paradigm [2, 3, 9] or by defining a whole new model [16]. While these developments were mostly motivated by the need to integrate continuous sensor feeds as inputs to workflows, we claim that some iterative scientific simulators like Gromacs [10] need a similar attention in order to be integrated in a scientific workflow. We also argue that a dataflow model of iterative components suits better the performance requirements of interactive visualization. In addition, the components may run at different rates. Simulations often run slowly while components handling control devices run very fast. The composition work consists then in focusing on inter-component communication and synchronization. It must guarantee that the whole constructed application remains fast enough to process interactions in a reasonable time so that the user can observe their effects on the simulation. In addition, the scientific nature of the applications requires a specific attention to data carrying. While some of the data can be dropped to speed the application up -e.g. the positions of the atoms in a molecular simulation has not necessarily to be frame-accurate at display-, the wholeness of other data may be essential to the coherence of the application. In [14], the authors stress the importance of well formalized coordination mechanisms in the coherence of simulations.

So the construction of a component-based real-time interactive application must deal with heterogeneous components and data and try to reach the best compromise between speed and coherence. The accessibility of the composition task must also remain a prerogative.

In this paper, we present a framework to specify dataflow component-based interactive scientific visualization applications. It consists in

- a component specification taking into account its iterative process,
- a composition model focusing on data communication and synchronization constraints,
- an automatic method to deduce inter-component communication and synchronization that fits the user's requirements at best and allows the components to run as fast as possible.

This paper is organized as follows: Section 2 introduces our component model and specifically its set of connectors. Section 3 explains the specific coordination challenges of real-time interactive scientific visualization and Section 4 our methodology to address them. Section 5 presents the results of our approach

applied to a real world application. In Section 6, we evaluate and situate our method among similar proposals and give the axes of our future work.

2 Component Model

We define a composition model to automatically build high-performance interactive applications. Its objectives are to

- formalize a component definition from an iterative process and heterogeneous code
- formalize an application construction based on additional elements to express inter component connections able to ensure coherence and performance.
- propose a composition model to automatically construct the intented applications.

2.1 Components

Unlike scientific computing pipelines, data-driven interactive visualization applications are meant to run continuously along with the simulation or interaction components at the heart of them. Looping is thus inherent to all of the components and can directly be encoded inside them juts like the *Push* behaviour described in [13]. A *component* encapsulates a specific task in the data processing pipeline. Formally, it is a quintuplet $A = (n, I, O, C, f)$ where n is the name of the component and I and O two sets of user defined input and output ports. I and O respectively include s (for *start*) and e (for *end*), two default triggering input and output ports. e, at the end of an iteration, emits a signal that can be used to trigger another object. s is the port that receives such signals. C are the coherence constraints of A. It is a set of disjoint subsets of $I - \{s\}$. Finally, f is a boolean to indicate that the component must run *freely*. We indeed distinguish special components called *interactive components* that collect user interactions. Typically, they can manage control devices. Specifying that a component is interactive (setting f to true) means that its iteration rate must only depend on its processing rate. This ensures that the component will not miss any interaction event. For a component A, $name(A)$ denotes its name, $I(A)$ and $O(A)$ its set of input and output ports respectively, $cstr(A)$ its set of coherence constraints, and $f(A)$ its type.

Components work iteratively. The *behavior* of the component A consists in: (1) waiting until all of its connected input ports are supplied with data and that s, if connected, receives a new signal (2) performing its computation task which can be a parallel task and (3) producing new data on all of its output ports and a signal on e. This process is called an *iteration* of A. Each component numbers its iterations. $i(A)$ is the *iteration rate* of the component. $i(A)$ depends, of course, on the time the component needs to perform its task but it could also depend on the time data takes to arrive to the input ports since A must wait until all of its connected input ports are fed. *input* and *output ports* are identified by a *name*.

Data circulating between ports are called *messages*. For a message m, $it(m)$ is the iteration number of the component that produced m. The components of our model can also handle empty messages, i.e. containing no data. This allows a component to go out of the waiting state as soon as all of its input ports are fed, even if not all with fresh data.

2.2 Application construction

Constructing an application consists in defining the components and the connectivity between their ports. To express this connectivity we define *Connectors* and *Links* in order to describe exactly how the data are transmitted from an emitter component to a receiver component.

Connectors Our component model adopts a set of *exogeneous connectors* [6] designed to support iterative components of different rates anywhere in the application graph. For example, with respect to recent approaches [6, 13, 16], it adds the ability to choose between blocking and non-blocking connections in order to let the end-user decide which processes should constantly be kept alive and which ones do not need to reprocess already processed data. Similarly, we introduce filtering connections comparable to the filtering nodes in [9] because they appear to be an essential alternative, for the sake of performance, to the systematic buffering of all the messages in use in [16]. On the other hand, we chose to not include explicit time parameters for activity triggering to keep our model the most generic possible. Connectors must be set between two components to determine the communication policy between them, i.e. the type of synchronization and the possibility to loose messages or not. A connector c is a quadruple $c = (n, \{s, i\}, \{o\}, t)$ where t is its type (see Figure 1) and i is an input port and o an output port. n and s are similar to their homonyms in the component. We use the same notations $name(c)$, $I(c)$, $O(c)$ and $type(c)$ as for components. c can contain several messages. When the sender writes a message on an output port, it simply adds this message to the connector and when the receiver reads its input ports, the connector delivers one of its messages.

Because the components might run at different rates, the connectors need to avoid the overflow of messages when the receiver is slower than the sender. On the other hand, the sender might also slow the receiver down if its iteration rate is lower. To tackle these problems, we propose five connection patterns besides the plain *FIFO*. *sFIFO*, *bBuffer* and *bGreedy* are similar to patterns described in [13].

- In a plain FIFO connection, the sender dispatches messages at its own rate without considering the receiver. To prevent overflows, this pattern adds a new condition to leave the waiting state. In addition to new input data, the sender must wait for a triggering signal usually sent by the receiver. This connector is called **sFIFO**, *s* standing for *synchronized*. However, as observed in [13], it can make slow components block their predecessors and,

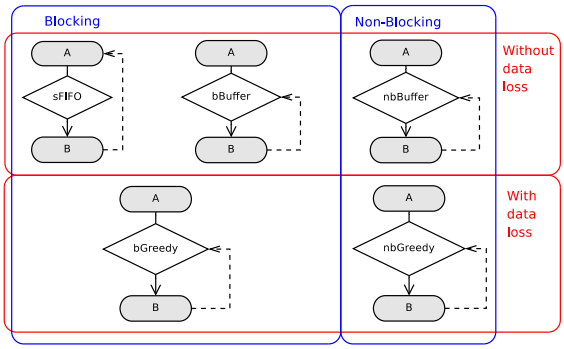

Fig. 1. The five connectors of our framework.

recursively, the entire application. This is particularly annoying in visualization applications where display components, that can be slow, are always at the end of the pipeline.

– Buffered FIFO connections can be useful to absorb overflows when one of the two components has an irregular iteration rate. When ready, the receiver triggers the sending of the oldest message in the buffer. We define the **bBuffer**, where *b* stands for *blocking*, because the absence of new message blocks the receiver. The **nbBuffer**, with *nb* standing for *non-blocking*, can, in contrast, dispatch empty messages when it is empty.
– A Greedy connector keeps only the last message provided by the sender and sends it upon the receiver's request. It is usually used to avoid overflows when it is not required that all messages are processed. The **bGreedy** and the **nbGreedy** are, respectively, the blocking and the non-blocking variants of this pattern.

Links Links connect components or connectors together through their ports. They are denoted by (x^p, y^q) with x, y component or connector, $p \in O(x)$ and $q \in I(y)$. There are two types of links:

– **A data link** transmits data messages. For a data link (x^p, y^q), we impose that $p \neq e$, $q \neq s$ and at least x or y is a connector. Indeed, as a connector is always required to define a communication policy, a data link cannot be directly set between two components.
– A **triggering link** transmits *triggering signals*. For such a link (x^p, y^q), we impose that x is a component, $p = e$ and $q = s$. The triggering links are illustrated by dashed lines in Figure 1. Please note that, to avoid deadlocks, neither components nor connectors wait for a triggering signal before their very first iteration.

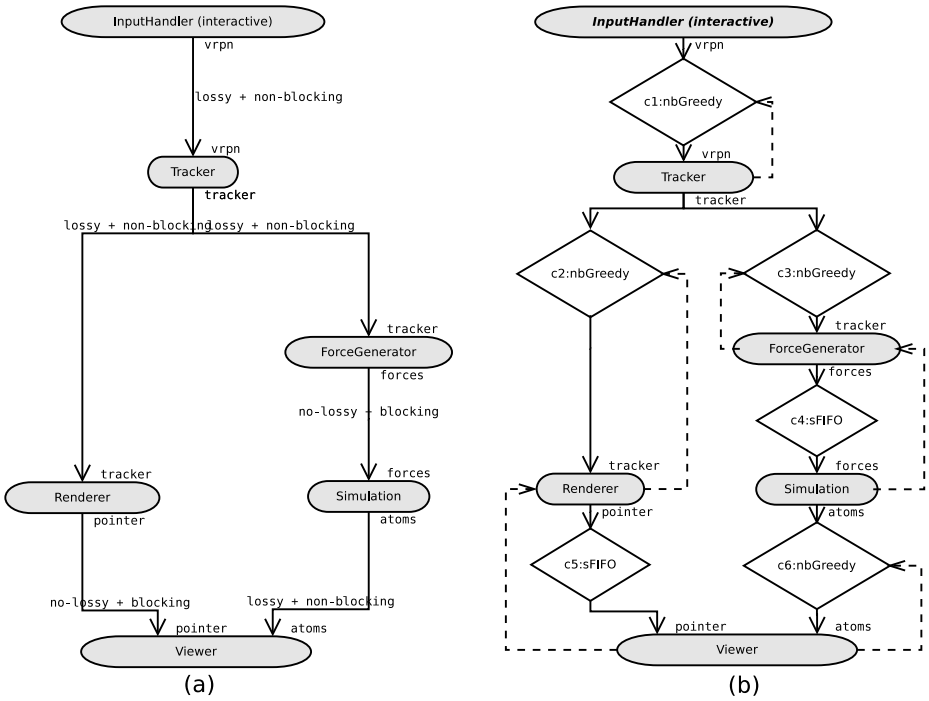

Fig. 2. (a) A specification graph (b) A corresponding application graph

2.3 Application graph

With these elements, the application construction can be represented by a graph called the *application graph*. The vertices of this graph are the components and the connectors. The edges represent the links.

Definition 1. *Let Cp be a set of components, Cn a set of connectors, Dl a set of data links and Tl a set of triggering links. The graph $App = (Cp \bigcup Cn, Dl \bigcup Tl)$ defines an* application graph*. In the remainder of this article, we call a* data path *of App an acyclic path in the graph $(Cp \bigcup Cn, Dl)$.*

Figure 2(b) illustrates the application graph of an interactive molecular dynamics application. An *InputHandler* communicates with a hardware controller like an Omni®. It transforms interaction events into streams that are filtered by the *Tracker* component to forward only the position of the pointer. This position is then used to compute, in real-time, the forces to be injected into the simulation in order to be applied to the molecule. It is also passed to a *Renderer* that computes the graphical representation of the pointer. We use an *nbGreedy* after the *InputHandler* because it iterates brokenly and with a high frequency. Then, while *ForceGenerator* and *Simulation* are synchronized thanks to blocking patterns, the *Viewer* is separated from them by an *nbGreedy* to obtain the

smoothest display possible. Building a whole application by putting together components, connectors and links is not easy for beginners, especially considering the specific requirements detailed in Section 3. In Section 4, we present a method to automate this construction.

3 Introducing Coherence

When building his application, the user makes local decisions by choosing the connectors to set between pairs of components. Controlling the coordination of the whole graph this way remains however a difficult task that can become unsolvable as the number of graph edges increases. Of course, one can set non-blocking connections everywhere to avoid slow downs. Nevertheless, this would lead to a rash global coordination which is contradictory with the precision expected from scientific applications. Alternatively, one can tie everything up with blocking and non-lossy patterns to ensure coherence but this would result in a general performance drop to the frequency of the slowest component. Our goal is to propose a composition model able to construct, from a user specification, an application ensuring performance and result reliability. To introduce our concepts, we first need a few preliminary definitions.

Definition 2. *Let App be an application graph. We call* segment *a data path of App. The starting vertex is called the* source*, denoted* $src(s)$*, and the arriving vertex is called the* destination*, denoted* $dest(s)$*. A message arriving at* $dest(s)$ *is called a* result *of* s*. The message from the source that originates this result is denoted by* $ori_s(r)$*. A segment whose source and destination are both components is called a* pipeline*.*

Definition 3. *Two pipelines* p_1 *and* p_2 *are* parallel *if and only if* $src(p_1) = src(p_2)$ *and* $dest(p_1) = dest(p_2)$ *and they do not share any other component.*

In interactive scientific visualization, the result reliability can be achieved, from a coordination point of view and apart from data type matters, by enforcing coherence constraints on the data streams incoming at a component from different input ports. In the application of Figure 2(b), the user might want the data issued by *Renderer* and *Simulation* to be synchronized at display, i.e. to come from the same iteration of the *Tracker*. More generally, the coherence between two input ports i_1 and i_2 means that if the two messages arriving at the same time at i_1 and i_2 are the results of two parallel pipelines starting at a single component A, then they are the "products" of the same iteration of A.

Definition 4. *Let* A *be a component and* i_1,$i_2 \in I(A)$*.* i_1 *and* i_2 *are said* coherent *if, for all pairs of parallel pipelines* (p_1, p_2) *such that the last edge of* p_1 *connects to* i_1 *and the last edge of* p_2 *connects to* i_2*, we can ensure* $it(ori_{p_1}(r_1)) = it(ori_{p_2}(r_2))$ *where* r_1 *and* r_2 *are two results of* p_1 *and* p_2 *read at the same iteration by* A*.*

In Figure 2(b), the coherence between the input ports *pointer* and *atoms* of the *Viewer* is not achieved. Indeed, as the *nbGreedy* connectors c2, c3 and c6 deliver only the last stored message and as the modules of the two pipelines run at different rates *Viewer* can receive data that are not issued from the same iteration of *Tracker*.

4 Automatic composition

This section describes how the user specifies the constraints on his application and the way we automatically build an application graph that ensures input port coherence while trying to preserve performance.

4.1 Application specification

The application specification helps the user focus on the expected properties of the communications in the application, avoiding technicalities. It is done by a directed graph called the *specification graph*.

Its vertices are the components of the application. Its edges, directed from the sender to the receiver, are labelled with the output and input ports and the constraints on the communications. These constraints are of two types: (1) the *message policy*, i.e. can this communication drop messages or not, and (2) the *synchronization policy*, i.e. should the receiver of the message be blocked when no new messages are available. Our aim is to compute an application graph that implements the specifications given by user following the overall rule *The generated application has to be, first of all, as safe as possible and then, as fast as possible*. The first step of the process consists in computing a preliminary application graph by replacing each edge of the specification graph with a connector following the rules of Table 1. As in many cases, several connectors fit the same combination, this table itself was created following the previous overall rule. The application graph of Figure 2(b) is obtained from the specification graph of Figure 2(a).

	Blocking policy		**Non-blocking or Interactive receiver**
Msg loss	bGreedy		nbGreedy
	Sender is interactive	**Sender not interactive**	
No msg loss	bBuffer	sFIFO	nbBuffer

Table 1. The choice of communication patterns

Next, the preliminary graph is transformed to implement first the coherence constraints and then to optimize the running time of the application. We first present how the coherence can be implemented in an application graph then we explain the different steps of the process.

4.2 Input port coherence

The coherence of two or more input ports of a component depends on the coherence of pairs of parallel pipelines that end at these input ports. The latter relies on a few basic concepts illustrated Figure 3.

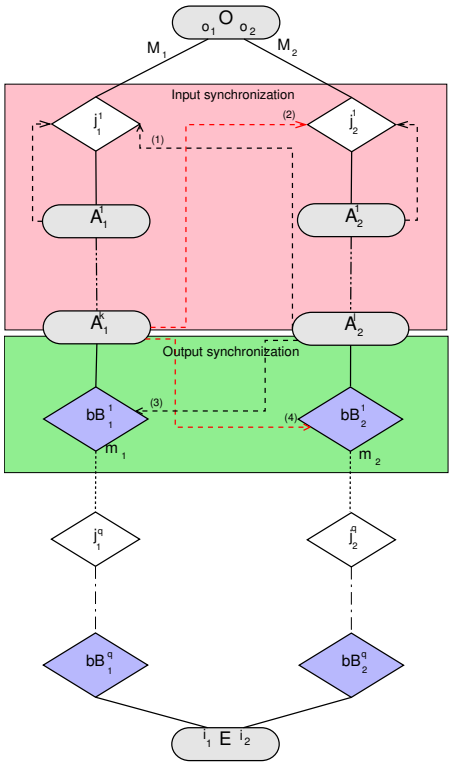

Fig. 3. Ensuring Coherence

Definition 5. *A segment* $(A_1, c_1, \ldots, A_{n-1}, c_{n-1}, A_n)$ *where* A_i $(1 \leq i \leq n)$ *is a component and* c_i $(1 \leq i \leq n-1)$ *is either a sFIFO or bBuffer connector is called a* synchronous segment.

Property 1. Let $s = (A_1, c_1, \ldots, A_{n-1}, c_{n-1}, A_n)$ be a synchronous segment and m_n a message produced by A_n, $it(m_n) = it(ori_s(m_n))$

The property is obvious since no message is lost inside a synchronous segment. Indeed A_n generates as many messages as A_1.

Definition 6. *A* junction *is a bGreedy, an nbGreedy or an nbBuffer connector between two consecutive synchronous segments making them* independent*.*

The more synchronous segments an application has, the faster it can run. However, the junctions between the segments of a pipeline are also the points where the coherence can be lost.

Definition 7. *In an application graph App, an* input synchronization *is a composition pattern involving*

- *two synchronous segments s_1, s_2 of respectively k and l components which are distinct and ended respectively by the components A_1^k and A_2^l,*
- *two junctions j_1, j_2 of the same type and preceding respectively s_1 and s_2,*
- *a* backward cross-triggering *consisting of (A_1^e, j_2^s) and (A_2^e, j_1^s).*

This pattern is denoted $J * (s_1, s_2)$.

Figure 3 illustrates the *input synchronization* where the junctions j_1^1 and j_2^1 are of same type and where the *backward cross-triggering* is represented by the two arrows labelled (1) and (2). This synchronization ensures that the junctions select their messages at the same time and that no new messages are accepted by the first components of the segments before all the components of the segments are ready for a new iteration. Note that this property is maintained when the two junctions are triggered by a same additional set of signals.

Definition 8. *In an application graph App, an* output synchronisation *is a composition pattern involving*

- *two synchronous segments s_1 and s_2 of repectively k and l components which are distinct and ended by components respectively A_1^k and A_2^l*
- *two bBuffer connectors bB_1 and bB_2 following respectively s_1 and s_2*
- *a* forward cross-triggering *consisting of $({A^k}_1{}^e, bB_2^s)$ and $({A_2}^l{}^e, bB_1^s)$.*

This pattern is denoted $(s_1, s_2) * bB$.

Figure 3 illustrates the *output synchronization* where the forward cross-triggering is represented by the arrows labelled (3) and (4). This composition pattern ensures that the delay between the synchronous segments to produce messages is absorbed. As the bBuffer connectors select their messages at the same time when all the last components of the synchronous segments are done, the messages are delivered also at the same time. Note that this property is maintained when the two bBuffer connectors are triggered by a same additional set of signals.

Definition 9. *In an application graph App, the composition $J * (s_1, s_2) * bB$ where s_1 and s_2 are two synchronous segments with no common components, is called a pair of* coherent *segments. $[J * (s_1, s_2) * bB]^q$ denotes the composition of q coherent segments $J^1 * (s_1^1, s_2^1) * bB^1 * \cdots * J^q * (s_1^q, s_2^q) * bB^q$.*

We denote M a series of messages, $|M|$ is the length of this series, and M^i denotes the i^{th} message of the series. A set of series of messages $\{M_1, \ldots, M_n\}$ are said *coherent* if $|M_1| = \cdots = |M_n|$ and $\forall j \in [1, |M_1|]$, $it(M_1^j) = \cdots = it(M_n^j)$.

Theorem 1. *Let App be an application graph and $(S_1, S_2) = J * (s_1, s_2) * bB$ a pair of coherent segments of App. If the series of messages M_1 and M_2 stored in the junctions j_1 and j_2 are coherent, then the set of messages m_1 and m_2 stored respectively in the bB_1 and bB_2 bBuffer connectors are such that $it(m_1) = it(m_2)$ and $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2))$ when the bBuffers are triggered.*

Proof. Since the series of messages stored in j_1 and j_2 are coherent and that they are triggered at the same time, respectively the messages M_1^i and M_2^i that they deliver have the same iteration number k_1. After this operation, the new series of messages stored in j_1, j_2 are still coherent.

By construction, the first components of the two segments begin a new iteration at the same time. So, their iteration numbers are always equal and denoted k_2. From Property 1, we know that the iteration number of each message delivered at the end of both segments, is equal to the iteration number of the message produced by the first components of the segments, i.e. k_2. Since the message m_1 stored in bB_1 and m_2 stored in bB_2 are made available only when the last components of s_1 and s_2 both finish their iterations, we do have that $it(m_1) = it(m_2) = k_2$ at this moment and $it(ori_{s_1}(m_1)) = it(ori_{s_2}(m_2))$.

Theorem 2. *Let App be an application graph and $(S_1, S_2) = [J * (s_1, s_2) * bB]^q$ two segments in App. If the series of messages M_1 and M_2 stored in the junctions j_1^1 and j_2^1 of the first coherent segments are coherent then the set of messages m_1 and m_2 stored respectively in the bB_1^q and bB_2^q bBuffer connectors of the last coherent segments are such that $it(m_1) = it(m_2)$ and $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2))$ when the bBuffers are triggered.*

Proof. According to Theorem 1 if the series of messages stored in the junctions are coherent then the messages delivered by the bBuffers connectors have the same iteration number. Therefore, the series of messages stored in the next junctions are coherent. An easy induction on q proves the theorem.

Figure 3 gives an example of two parallel pipelines p_1 and p_2 that ensure the coherence of two ports. This application graph is composed of an initial component O (for origin), of a composition pattern $[J * (s_1, s_2) * bB]^q$ such that $(O^{o_1}, j_1^{1^i})$ and $(O^{o_2}, j_2^{1^i})$ and of an final component E such that $(Bb_1^{q^o}, E^{i_1}$ and $(Bb_2^{q^i}, E^{i_2})$. The junctions are such that j_2^j and j_1^j $1 \leq j \leq q$ are of the same type. If M_1 and M_2 denote the series of messages coming from the output ports of the component O, M_1 and M_2 are coherent. So the messages stored in the first junctions j_1^1 and j_1^2 are coherent. According to Theorem 2 for the messages delivered by the bB_1^q and the Bb_2^q bBuffers $it(m_1) = it(m_2)$. Moreover if S_1 denotes the data path from j_1^1 to bB_1^q and S_2 the one from j_2^1 to bB_2^q then $it(ori_{S_1}(m_1)) = it(ori_{S_2}(m_2))$. This means that $ori_{S_1}(m_1)$ and $ori_{S_2}(m_2)$

come from the same iteration of O hence $it(ori_{p_1}(r_1)) = it(ori_{p_2}(r_2))$ which corresponds to the definition of the coherence.

To construct an application graph under coherence contraints, we propose to transform our preliminary application graph into a new one such that the data paths implied in a coherence constraints are composed of coherent segments.

4.3 Transformations for coherence construction

The different steps of our construction are illustrated by the specification graph of Figure 2(a) where the user wants the coherence between ports *pointer* and *atoms* of component *Viewer*. Figure 2(b) gives the preliminary application graph of this application.

Coherence graphs The first step of the transformation consists in looking for parallel pipelines that must be coherent. They are collected in coherence graphs.

Definition 10. *Given an application graph G and a coherence constraint $\mathcal{C}$ of the component A in G, the* coherence graph *of $\mathcal{C}$ in G is the subgraph of G that contains all the parallel pipelines of G with the members of $\mathcal{C}$ as destinations.*

Our example application has one coherence constraint $\{pointer, atoms\}$. Its coherence graph is inside the frame in Figure 4(a).

Path segmentation As seen in Section 4.2, our construction is based on parallel pipelines which have the same number of independent segments. So, the purpose of this step is to create, if necessary, new segments, i.e. switch some connectors from {sFIFO or bBuffer} to nbBuffer, or from {bGreedy or nbGreedy} to {sFIFO, bBuffer or nbBuffer}. This is possible because we allow the system to relax blocking, non-blocking or lossy constraints of the specification graph. In contrast, no-lossy constraints are never relaxed.

Instead of making coherent each pair of parallel pipelines, the process is done on the whole application in the same time. This allow us to take into account all the constraints and avoid backtrackings in the process. For that, we use a linear system where each variable is associated to a connector. The domain of the variables is $\{0, 1\}$. 0 means that the connector is either a sFIFO or a bBuffer, and 1 any of the three other patterns. Since these three other patterns define junctions, it is sufficient to impose that the sums of the variables of the parallel pipelines are equal to ensure that they have the same number of segments.

Formally, let *App* be an application graph and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ the set of coherence constraints of the modules in *App*. We denote $G|_{\mathcal{C}_i}$ the coherence graph of $\mathcal{C}_i$ in *App*. For each component A of *App*, we denote $PP_A(App|_{\mathcal{C}_i})$ the set of parallel pipelines coming from A and leading to $\mathcal{C}_i$. For each connector c of *App*, we associate the variable v_c of domain $\{0, 1\}$. For each path p in *App* that contains the sequence of connectors $c_1, \ldots, c_k$, we define $Sum(p) = v_{c_1} + \cdots + v_{c_k}$. For each component A, and each of its constraints $\mathcal{C}$, we define the set of equations $Eq(PP_A(App|_{\mathcal{C}})) = \{Sum(p_1) = \cdots = Sum(p_l) | \, p_j \in PP_A(App|_{\mathcal{C}}), 1 \leq$

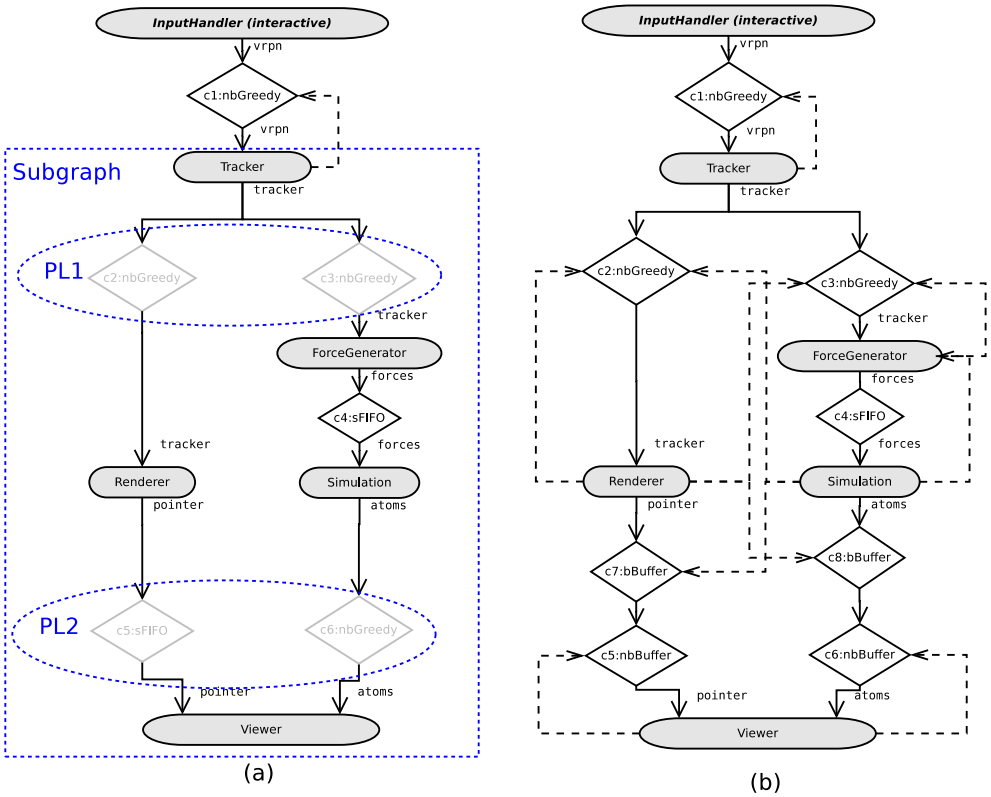

Fig. 4. Application graph before (a) and after (b) equalization and synchronization

$j \leq l\}$. The set of equations corresponding to the problem is then $Eq_{App} = \bigcup\{Eq(PP_A(\mathcal{C})|A \in App, \mathcal{C} \in cstr(A)\}$.

Additional constraints are also added to the problem to avoid misleading solutions. For each connector c of *App*, according to the properties of the corresponding connection in the specification graph and those of the sender and the receiver, we determine the set of compatible patterns. If this set contains only elements of {nbBuffer, bGreedy, nbGreedy}, we add $v_c = 1$ to the linear system. The set of these additional equations is denoted Fix_{App}. Most of the time, the system has many solutions that are not equivalent from a performance point of view. We give then priority to those that maximize the application's performance, i.e. that preserve at best the initial junctions. This is expressed by the following objective function $Maximize(Sum(J_{App}))$ where J_{App} is the set of junctions initially set in *App* and $Sum(J_{App}) = \Sigma_{c \in J_{App}}(v_c)$. So the linear problem we want to solve is $Eq_{App} \bigcup Fix_{App} \bigcup Maximize(Sum(J_{App}))$.

For the application of Figure 4(a), this process produces the following problem $\{c_2 + c_5 = c_3 + c_4 + c_6\} \bigcup \{\emptyset\} \bigcup Maximize(c_2 + c_3 + c_6)$ and its solution is $\{c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$.

Plateau equalization It remains now to definitively set the pattern of each junction. For that, we define the notion of *plateau*.

Definition 11. *Let App be an application graph,* $O * [J * (s_1, s_2) * bB]^q * E$ *two parallel pipelines whose last edges connect to two ports of the same coherence constraint of* E*. We say that the junctions* j_1^i *and* j_2^i *(*$i \in [1, q]$*) are of the* same level*, which is denoted* $j_1^i \leftrightarrow j_2^i$*. The reflexive-transitive closure of* $\leftrightarrow$ *is denoted* $\leftrightarrow^*$*. A* plateau *is the set of the junctions of the same equivalence class of* $\leftrightarrow^*$*.*

It can be proved that two different junctions j_l^i and j_l^k that belong to the same pipeline involved in at least one coherence constraint cannot be in the same plateau. This avoids interblocking input-output synchronizations.

The connectors of a given plateau must be of the same type to ensure a coherence constraint. This is the object of the next step. To solve the problem on the whole graph, we regroup all the plateaus that have connectors in common into a single subset of connectors. When a subset contains connectors of different types, we set all the connectors of the subset to nbBuffer if the subset contains at least a nbBuffer pattern and a nbGreedy otherwise. Figure 4(a) shows that our example application has two plateaus *PL1* and *PL2*. *PL1* will be kept as it is because the connectors are already of the same type. In *PL2*, the non-lossy constraint on c5 enforces *nbBuffer* as the only option.

At the end of this step, we first add the backward cross-triggerings to the junctions. Since a plateau may involve more than two segments, our construction generalizes Definition 7. For a plateau $j_1, \ldots, j_n$ and the segments $s_1, \ldots, s_n$ ending with components $A_1, \ldots, A_n$ we add the set of edges $\{(A_i^e, J_j^s) | i \neq j\}$. Moreover to implement the output synchronization, we add one bBuffer connector just after each A_i ($i \in [1, n]$) and add the edges for the forward cross-triggerings. This ensure the required coherences.

Finalization and optimization To finalize the application graph, our system adds all the missing trigger links, factorizing those that can be. For example, in the application graph in Figure 4(b), *Simulation* triggers *ForceGenerator* which triggers c3. This boils down to *Simulation* triggering c3. Finally, the system outputs an XML file that can be used to generate the concrete application.

5 Experimental results

To validate our component-based model, we use FlowVR [1], a middleware to develop and run high-performance interactive applications. It is based on a component called *module* which consists in an iterative process with input and output ports and can model our component. It also offers an assembly model expressed as a communication network from connection elements covering standard communication or synchronization patterns. From specific needs, it is also possible to developp ad hoc connection elements and, in particular, our connector elements have been developed and integrated into FlowVR.

Concerning our automatic application construction, we implemented a software prototype that parses a user specification XML file and generates the adequate application graph according to the given information and constraints. This application graph is then transformed into a FlowVR assembly model which can be processed by FlowVR to execute the application on a cluster architecture.

We have been testing our software prototype on the example application used throughout this paper. This application is representative as it contains connections with different arities and a coherence constraint between the input ports *pointer* and *atoms* of the display component. The simulation component in the implementation is run by Gromacs, a very popular molecular dynamics engine and the other components have been specifically implemented. The tests were run with a 3000 atom molecule on a 2x2.5Ghz CPU.

We tested several versions of the graph, including one with only bBuffers and sFIFOs as we suppose it would be set by hand by someone unfamiliar with the connection patterns and unsure of the impact of message dropping on the coherence. As this graph has only one-segment pipelines, it doesn't need any specific coherence processing. However, the overall performance drops to the iteration rate of the slowest component, i.e. the *Viewer* in this case, so around 14.8 it/sec. In constrast, the implementation of the automatically generated application graph of Figure 4(b) keeps the display component loosely connected to the rest and the simulation can run at up to 154.2 it/sec. The coherence between the input ports *atoms* and *pointer* is also maintained.

6 Discussion and future work

Coherence - along with performance and simplicity - is a major criteria for scientists when building their own software. To our knowledge, coherence as we mean it in this paper and its automatic fulfillment by graph modification has not been explored yet. We proved that a complex application can be automatically constructed from a user specification expressed in terms of communication and coherence constraints. Not only do the generated applications ensure the coherence of data input wherever it is requested but they also guarantee the safest -in terms of overflows or unwanted output overwriting- and the fastest execution possible. A software prototype used on a real world scientific application validates our approach.

The focus of our future research will be to enrich our model with component hierarchy and define a coherence that supports not only regular message streaming but also event-based message emission. Moreover we plan to enrich the coherence model. For example, the user may want to specify coherence constraints such as allowing a variation of n iterations between two components or specifying coherence wrt messages produced by two different components. We also plan to implement a composition UI and deal with data type compatibility and adaptation still in the perspective of simplifying the assembly of heterogeneous software components.

References

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. *LNCS*, pages:497–505, 2004.
2. D. Barseghian, I. Altintas, M.-B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E.-T. Borer, and E.-W. Seabloom. Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50, January 2010.
3. Biörn Johan Bkörnstad. *A Workflow Approach to Stream Processing.* PhD thesis, Zürich, 2007.
4. S.P. Callahan, J. Freire, E. Santos, C.E. Scheidegger, C.T. Silva, and H.T. Vo. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 747. ACM, 2006.
5. O. Delalande, N. Férey, G. Grasseau, and M. Baaden. Complex molecular assemblies at hand via interactive simulations. *Journal of Computational Chemistry*, 30(15), 2009.
6. P. Velasco Elizondo and K.-K Lau. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 83(7):1165–1178, 2010.
7. Y. Gil, V. Ratnakar, J. Kim, P. Gonzalez-Calero, P. Groth, J. Moody, and E. Deelman. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 99, 2010.
8. 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*, pages 1–31, 2002.
9. Chathura Herath and Beth Plale. Streamflow Programming Model for Data Streaming in Scientific Workflows. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 302–311, May 2010.
10. B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, mars 2008.
11. Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(Web Server issue):W729–32, juillet 2006.
12. 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.
13. C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Proceedings of the Workshop on Workflows in Support of Large-Scale Science*, page 19–23, 2006.
14. J. Siebert, L. Ciarletta, and V. Chevrier. Agents & artefacts for multiple models coordination. *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 20–24, 2010.
15. Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, January 2006.
16. A. Wombacher. Data Workflow-A Workflow Model for Continuous Data Processing. *Data Processing*, 2010.
17. Z. Zhao, A. Belloum, A. Wibisono, F. Terpstra, P.T. de Boer, P. Sloot, and B. Hertzberger. Scientific workflow management: between generality and applicability. In *Quality Software (QSIC 2005).*, pages 357–364. IEEE, 2006.