Supervision et vérification décentralisées de chorégraphies de services

Aymen Baouab, Olivier Perrin, Claude Godart

Supervision et vérification décentralisées de chorégraphies de services

Aymen Baouab, Olivier Perrin, Claude Godart

LORIA – INRIA Nancy – Université de Lorraine
F-54500 Vandœuvre-lès-Nancy, France
{aymen.baouab,olivier.perrin,claude.godart}@loria.fr

RÉSUMÉ. *Les chorégraphies de services sont de plus en plus adoptées comme un moyen de collaboration entre des organisations partenaires. Cependant, la quasi-totalité des approches proposées pour le suivi des compositions de services sont limitées à une même zone de confiance et ne peuvent donc pas être adaptées pour superviser les échanges de messages des processus métiers qui s'étendent à travers de nombreuses organisations administrativement indépendantes. Dans cet article, nous proposons une approche décentralisée et événementielle pour la supervision et la vérification des interactions d'une chorégraphie inter-organisationelle. Des évènements sont définis et traités par des unités de supervision implantées le long des frontières de chaque organisation et invoqués à chaque détection d'un nouveau message échangé avec une organisation partenaire. Après la vérification de chaque message par rapport au schéma de la chorégraphie, une notification est automatiquement générée, envoyée et propagée à un sous ensemble précalculé de participants. En cas de violation, un algorithme de recherche de la cause originelle est proposé.*

ABSTRACT. *Cross-organizational choreographies are increasingly adopted by different companies. In order to guarantee that all involved partners are informed about errors that may happen in the collaboration, it is necessary to monitor the execution process by continuously observing and checking message exchanges during runtime. This allows a global process tracking and evaluation of process metrics. In this paper, we present an approach for decentralized monitoring of cross-organizational service-based processes. We define an hierarchical propagation model for exchanging external notifications between the collaborating parties in order to limit data exposure and avoid bandwidth overhead. The generation of notifications relies on state change of choreographies allowing to transparently observe the external behavior of each participant without providing information about the internal processes. The collected monitoring data can be further used for the evaluation of global process metrics by expressing and evaluating statistical queries over execution traces. In case of violation, an algorithm is proposed to find the root cause.*

MOTS-CLÉS : *décentralisation, service, chorégraphie, supervision, vérification, violations.*

KEYWORDS: *decentralization, service, choreography, monitoring, verification, root cause.*

Table des matières

1. Introduction

De nos jours, la croissance continuelle des entreprises incite celles-ci à externaliser et à sous traiter certaines de ses activités. Cela a conduit à une forte augmentation du nombre d'interactions entre les différents partenaires en collaboration. Pour être capable d'externaliser leurs opérations ou vendre une partie de leurs processus métiers, certaines entreprises éprouvent le besoin de partitionner leurs processus et de séparer les éléments qui les constituent (Wetzstein *et al.*, 2010). Une chorégraphie inter-organisationnelle peut être mise en œuvre afin de permettre la coordination et la synchronisation de ces différentes partitions. Ainsi, chaque organisation est en charge d'exécuter une partie du processus global. Parmi les inconvénients majeurs de la sous-traitance, nous pouvons citer la perte de maîtrise, le manque de flexibilité de la part du prestataire, le risque de délais trop longs et le manque d'information et de transparence. Afin de pallier ces problèmes, il est nécessaire pour chaque organisation de superviser ses fragments externalisés, et ce, de façon abstraite sans pour autant dévoiler la logique métier de chaque partenaire. En effet, la propagation des données de supervision entre les partenaires permet une gestion décentralisée des informations recueillies, et ce, dans le but d'offrir plus de maîtrise au niveau de chaque organisation, et de réduire les délais et les coûts en cas d'occurrence d'exceptions.

Durant la dernière décennie, la supervision des compositions de services a été l'objet de plusieurs travaux de recherche (Ardissono *et al.*, 2008 ; Francalanza *et al.*, 2010 ; Moser *et al.*, 2010 ; Chafle *et al.*, 2004 ; Halle, Villemaire, 2009). Néanmoins, les approches proposées sont limitées aux compositions regroupant des services d'une même zone de confiance et ne peuvent pas être adaptées aux chaînes de production et d'approvisionnement qui s'étendent à travers de nombreuses organisations administrativement indépendantes. Ainsi, il est nécessaire de trouver un moyen efficace pour permettre une propagation instantanée des exceptions internes à travers les frontières organisationnelles. Dans ce sens, nous avons proposé un modèle architectural et un mécanisme de supervision et d'échange de notifications (Baouab *et al.*, 2011 ; Baouab, Fdhila *et al.*, 2012 ; Baouab, Perrin, Godart, 2012). Nous proposons une approche qui permet une supervision décentralisée de modèles de chorégraphies. Dans notre approche, les échanges inter-organisationels de messages sont perçus comme des évènements qui sont traités par des unités de supervision implantées dans chaque organisation et invoqués à chaque détection d'un nouveau message échangé avec une organisation partenaire. Après la vérification de chaque message par rapport au schéma de la chorégraphie, une notification est automatiquement générée, envoyée et propagée à un sous ensemble pré-calculé de participants. Par rapport à (Baouab *et al.*, 2013), nous présentons ici la mise en œuvre de la supervision, ainsi que la façon dont sont détectées les violations, en insistant sur l'amélioration apportée par rapport à des approches similaires, en particulier grâce à la définition d'anti-patrons et au mécanisme d'agrégation des violations qui permet de remonter plus rapidement à la source du problème (minimisation importante du nombre de messages à traiter) .

La section 2 montre l'intérêt de notre approche sur un exemple de chorégraphie. La section 3 présente l'approche de classification hiérarchique des participants. Ensuite,

la section 4 décrit les algorithmes de configuration et d'exécution. La section 5 montre comment mettre en œuvre notre approche de supervision, ainsi que les apports par rapport aux approches existantes. La section 6 décrit la détection des violations, et la possibilité d'agréger est définie dans la section 7. La section 8 présente quelques travaux proches, et la section 9 conclut cet article.

2. Chaînes d'approvisionnement et suivi des processus inter-entreprises

Pour illustrer les concepts et la démarche de l'approche présentée dans cet article et pour bien comprendre les problèmes qui peuvent exister, nous adoptons le scénario d'une chorégraphie inter-organisationnelle qui englobe trois types de participants (si on ne compte pas le client final) jouant les trois rôles suivants : Revendeur, Fournisseur et Constructeur dans une chaîne d'approvisionnement. Dans ce type de chaîne inter-organisationnelle, un revendeur peut interagir avec plusieurs fournisseurs qui, à leur tour, interagissent avec plusieurs constructeurs dans le but d'obtenir un devis pour des biens ou des services. La *figure* 1 montre un exemple de chorégraphie impliquant sept organisations principales: un client (C), un revendeur (R), deux fournisseurs (SA et SB), et trois constructeurs ($A1$, $A2$, et $A3$).

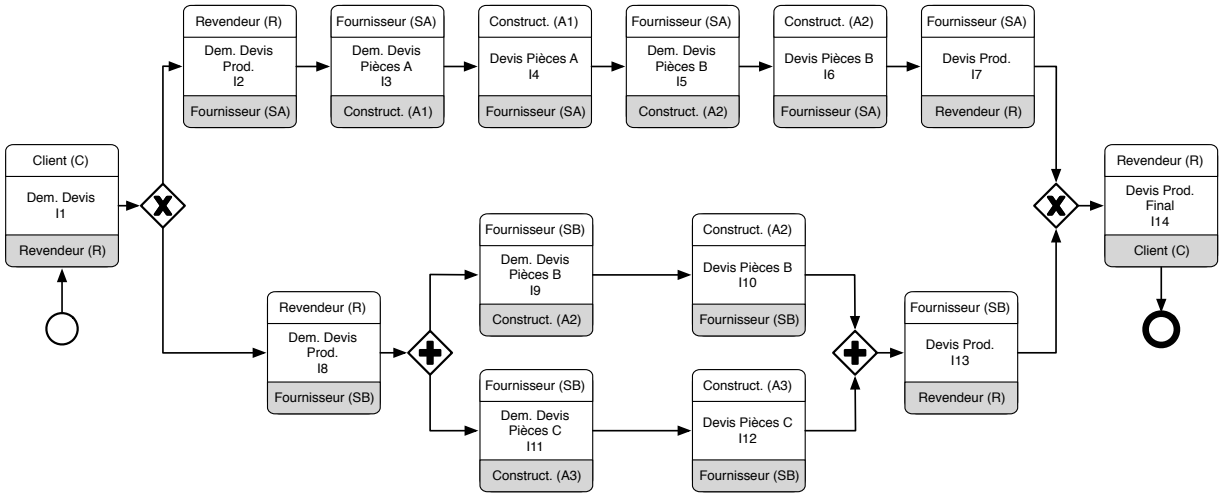


Figure 1. Diagramme d'interaction d'une chaîne d'approvisionnement (BPMN 2.0)

Dans cette chorégraphie, 14 interactions bi-partenaires sont définies ($I_{1,...,14}$). Dans un premier temps, le revendeur reçoit une demande de devis de la part du client (I_1). Ensuite, il sélectionne un des deux fournisseurs (suivant le contenu de la demande) et élabore une nouvelle demande. En cas de sélection du fournisseur SA (à travers I_2), les constructeurs $A1$ et $A2$ seront invoqués en séquence ($I_{3,...,6}$). Dans l'autre cas (i.e. SB sélectionné), les constructeurs $A2$ et $A3$ seront invoqués en parallèle ($I_{9,...,12}$). Après avoir retourné les résultats finaux (I_7 ou bien I_{13}), le revendeur pourra enfin répondre au client par un devis complet (I_{14}). La chorégraphie ainsi présentée permet d'offrir au client final un service de demande de devis préalable. Dans le cas d'une chaîne de fabrication sur mesure de pièces originales, l'exécution d'une telle chorégraphie peut prendre des jours (voire des semaines pour des produits complexes nécessitant des pièces particulières).

Dans un cadre inter-organisationnel, chaque organisation n'est en relation qu'avec les partenaires qui lui sont directement connectés. Techniquement, un service (ou un sous-processus exposé en tant que service) interagit en invoquant ses fournisseurs et en étant invoqué par ses consommateurs avec lesquels il a déjà conclu des accords de niveau de service. Lors de l'exécution, il n'a donc aucune information sur le reste des participants de la chorégraphie. Toutefois, un service dépend de tous ses services clients, que ce soit par un lien direct ou transitif. Toute organisation peut donc s'intéresser à la surveillance de tous les fragments de processus externalisés et/ou exécutés par ses sous traitants. Dans notre exemple, le client C interagit uniquement avec le revendeur R et ne peut donc pas voir ce qui se passe au-delà. Après avoir envoyé sa demande à R, C restera en attente de la réponse sans avoir aucune information sur l'état actuel de sa demande. Ainsi, il serait intéressant de garder C informé de l'évolution de l'exécution de la chorégraphie. Un tel exemple montre la nécessité d'échanger des notifications afin de permettre le suivi d'une chorégraphie. En effet, dans une configuration centralisée (e.g. orchestration), ce problème ne se pose pas puisque tout est coordonné par une unité centrale qui peut connaître à tout moment l'état actuel de l'exécution. Par contre, lorsqu'il s'agit d'une chorégraphie qui franchit les frontières de plusieurs organisations administrativement indépendantes, la tâche s'avère délicate. En outre, l'échange des notifications permet d'avoir une vue plus large que la vue locale de chaque participant et de recueillir des données supplémentaires qui pourraient être utilisées a posteriori pour mesurer la performance globale du processus et le suivi de la réalisation des objectifs de l'entreprise.

3. Mécanisme décentralisé pour l'échange de notifications entre partenaires

Un superviseur traditionnel est censé superviser la conformité de l'exécution des processus locaux par rapport aux modèles définis et aux règles métiers de sa propre organisation administrativement indépendante (i.e. en se référant uniquement à ce qu'il peut voir localement). Ce type de supervision ne permet donc pas le suivi global d'une chorégraphie inter-organisationnelle puisqu'il n'est pas possible de voir ce qui se passe au-delà des frontières de l'organisation (e.g. les communications entre d'autres partenaires de collaboration). Pour superviser l'ensemble de la chorégraphie, des solutions ont été proposées (Ardissono *et al.*, 2008 ; Chafle *et al.*, 2004 ; Wetzstein *et al.*, 2010). Ces approches sont centralisées et se basent donc sur un superviseur central implanté dans une organisation à laquelle tous les autres participants doivent faire confiance. Cette entité centrale est notifiée par chaque participant chaque fois qu'un événement se produit, et ce, en utilisant, par exemple, le mécanisme de « $publish/subscribe$ » ou publier/ s'abonner (Wetzstein *et al.*, 2010) . Néanmoins, elle peut présenter un point de défaillance unique et peut ne pas être adaptée lorsqu'il n'y a pas d'entité de confiance commune à tous les participants (e.g. cas d'une chaîne d'approvisionnement avec plusieurs pays). En plus, on pourra aussi citer tous les autres inconvénients d'une configuration centralisée (e.g. goulot d'étranglement).

Pour faire face à ce genre de problème, nous proposons un mécanisme automatisé et décentralisé pour l'échange de données de supervision entre les participants.

Nous définissons de nouveaux canaux pour l'échange de notifications entre les différents participants. Notre approche n'est pas intrusive, c'est-à-dire que ces canaux nouvellement définis sont indépendants des canaux dans lesquels les messages de la chorégraphie sont échangés. Une unité de supervision locale, appelée EFM, peut voir « passivement » ce qui est échangé sans rien modifier dans le modèle de chorégraphie. La *Figure* 2 montre un aperçu sur notre approche.

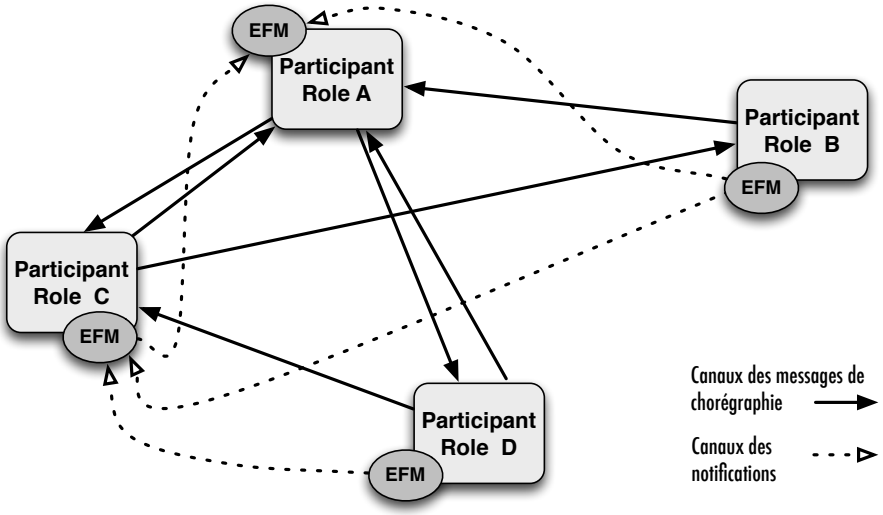


Figure 2. Mécanisme décentralisé pour l'échange de notifications entre partenaires

3.1. Définition des notifications externes

Lors de l'exécution, le modèle de chorégraphie est utilisé comme une base pour observer le comportement des participants. Notre approche repose uniquement sur les changements d'états de la chorégraphie (i.e. quand un message est envoyé ou reçu) permettant ainsi le suivi des échanges de messages d'une manière discrète (i.e. les messages échangés ne sont pas modifiés par les EFMs et les services composant la chorégraphie de base ne sont pas conscients des EFMs et des notifications créées et échangées). Chaque notification est corrélée à un message défini dans le modèle de la chorégraphie. Nous définissons une notification comme suit.

DÉFINITION 1 (Notification). — *Une notification $n \in \mathcal{N}$ est un tuple (c_i,t,s,d,m_t) avec c_i l'identifiant de l'instance de chorégraphie (utilisé pour la corrélation), t le timestamp de la génération de l'évènement, s,$d \in \mathcal{P}$ (respectivement, la source et la destination du message associé), et $m_t \in \mathcal{M}_\mathcal{T}$ (le type/structure de message).*

3.2. Classification hiérarchique des partenaires

Dans un souci de réduire le nombre de notifications échangées, nous avons choisi un échange sélectif. En d'autres termes, nous essayons de réduire le nombre de partenaires qui doivent être notifiés. Pour cela, nous introduisons un nouveau type de relation entre les participants de la chorégraphie.

Afin de permettre l'envoi sélectif des notifications, il faut commencer par déterminer le sous-ensemble des partenaires potentiellement intéressés par l'événement correspondant. Dans le cas des chaînes d'approvisionnement, les partenaires intéressés sont ceux qui sont considérés comme des consommateurs ou clients de ce participant (e.g. une entreprise est considérée comme cliente chez ses sous-traitants et est donc intéressée par l'accomplissement de toutes ses parties métiers externalisées). Nous procédons à une classification hiérarchique des participants suivant la relation producteur/consommateur et l'évolution de la valeur ajoutée dans une chaîne d'approvisionnement de services donnée. Nous commençons par mettre le client (i.e. l'initiateur de la chorégraphie) au plus haut dans la hiérarchie (sur la couche supérieure). Les couches les plus basses sont constituées d'un ensemble de fournisseurs de services. Ainsi, chaque participant d'une couche donnée fournit un service à un service de la couche qui est juste au dessus (voir *figure* 3). Comme chaque couche dépend (directement ou transitivement) de toutes les couches inférieures, tout participant qui attend un service de la couche inférieure sera intéressé par le suivi de son exécution. Nous autorisons ainsi la propagation des notifications uniquement du bas vers le haut de la hiérarchie. Pour permettre une telle classification, nous introduisons la notion de *super / sous-partenaire* comme suit.

DÉFINITION 2 (Super / sous partenaire). — *Un participant $P_i \in \mathcal{P}$ est appelé le $super-partenaire$ d'un participant P_j ssi P_i est l'émetteur dans la première interaction définie dans la vue locale de P_j, ou encore, l'instance du sous-processus de P_j est créée suite à un message venant de P_i. En d'autres termes, P_i est responsable de la participation de P_j dans la chorégraphie. Ainsi, P_j est appelé $sous-partenaire$ de P_i. Formellement, on note :*

$$Super(P_i) = P_j \Leftrightarrow P_i \in Sub(P_j)$$

Dans notre exemple de chaîne d'approvisionnement (présenté précédemment dans *Figure* 1) : $Super(SA) = R$; $Super(SB) = R = Super(Super(A2))$; $SA, SB \in Sub(R)$; $A1, A2 \in Sub(SA)$; $A2, A3 \in Sub(SB)$. Il faut remarquer que notre approche assume que la chorégraphie soit réalisable (Lohmann, Wolf, 2009) et avec un seul initiateur. Généralement, dans une chaîne d'approvisionnement, un fournisseur de services peut avoir, lui même, un ou plusieurs fournisseurs, qui eux aussi, peuvent avoir, à leur tour, d'autres fournisseurs et/ou sous-traitants, etc. Ceci fait qu'une telle structure est hiérarchique (Haq *et al.*, 2009). De cette façon, chaque participant aura exactement un *super-partenaire* et peut avoir un ou plusieurs *sous-partenaires*. En tant que tel, les participants peuvent être classés dans une arborescence avec l'initiateur comme racine.

Chaque participant notifie son *super-partenaire* en générant une nouvelle notification chaque fois qu'il échange un message (envoi ou réception). En plus des notifications qu'il génère, il lui transmet aussi toutes celles reçues de la part de ses *sous-partenaires*, et ce, directement dès qu'il les reçoit. Ainsi, les notifications se propagent toujours du bas vers le haut de l'arborescence des partenaires. L'utilisation de cette structure hiérarchique permet de réduire le nombre de notifications échangées. Ceci permet aussi de recevoir chaque notification au maximum une fois et élimine donc

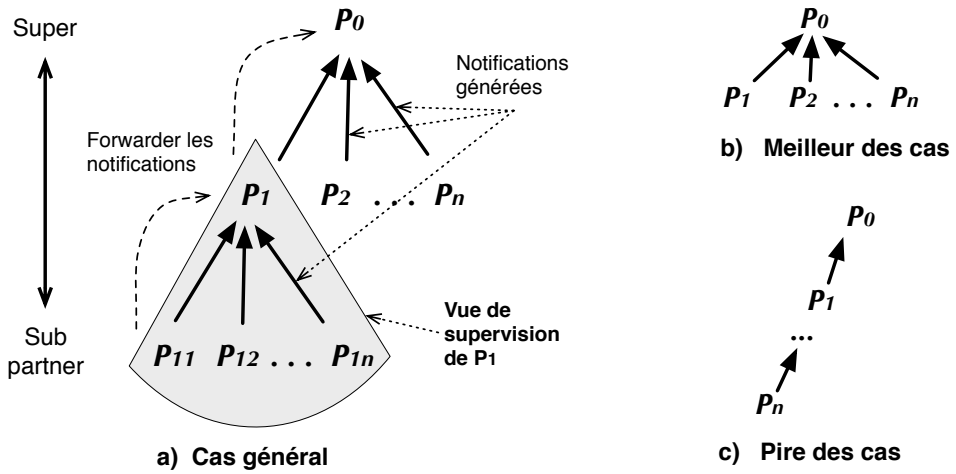


Figure 3. Classification hiérarchique des partenaires

toute redondance suite à une réception multiple des notifications sur le même évènement de la part de deux sources différentes.

Nous avons montré dans (Baouab *et al.*, 2013) que la complexité de notre approche par rapport au nombre total de notifications échangées était en moyenne en $O(p.Log(p))$ notifications (avec p le nombre de participants). Dans le meilleur des cas (lorsque la profondeur est égale à 1), la complexité est linéaire (voir *figure* 3.b). Dans le pire des cas et lorsque l'arbre est totalement déséquilibré (voir *figure* 3.c.), nous avons besoin de $O(p^2)$ notifications. Ici, l'arbre ressemble à une liste chaînée (aussi appelé arbre dégénéré).

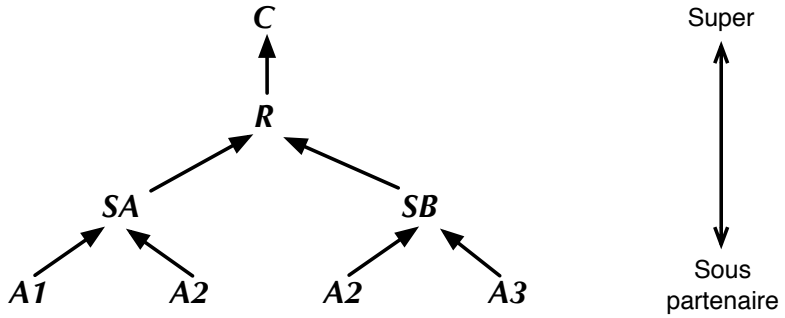


Figure 4. Arborescence des partenaires pour l'exemple de motivation

La *figure* 4 montre l'arbre généré pour notre exemple de chaîne d'approvisionnement. Nous remarquons que le participant $A2$ figure deux fois dans l'arbre. En effet, suivant le chemin emprunté dans le schéma de la chorégraphie (i.e. le choix du fournisseur entre SA et SB fait par le revendeur R), le constructeur $A2$ pourra avoir un des deux fournisseurs comme *super-partenaire*. Nous rappelons que l'affectation des *super-partenaires* se fait au cours de l'exécution de la chorégraphie au moment de la réception du premier message pour chaque participant et ce de façon indépendante pour chaque instance. Donc, ici $A2$ aura, à chaque instance, SA ou bien SB comme *super-partenaire* et non pas les deux. En d'autre termes, le *super-partenaire* de chaque participant est toujours unique. Ceci permet, entre autres, de garantir à notre classification de partenaires de garder sa structure hiérarchique.

3.3. Vue de supervision externe (EFM-View)

Une vue locale dans une chorégraphie représente la visibilité du point de vue d'un des participants. Elle fait référence au modèle restreint à toutes les interactions d'un seul partenaire. Par défaut, chaque participant est limité seulement à sa propre vue. En utilisant notre approche hiérarchique d'échange de notifications, chaque EFM reçoit des notifications sur tout changement d'état relatif à l'ensemble de ses *sous-partenaires*. Ainsi, l'EFM de chaque participant aura une vue plus large que la vue locale de ce dernier. Nous appelons cette vue nouvellement créée « vue de supervision externe » (*EFM-View*). L'*EFM-View* d'un participant P_i inclut toutes les interactions ayant comme émetteur ou receveur un des *sous-partenaires* de P_i. Cette vue comprend également l'ensemble des contraintes sur le séquencement de ces interactions. Formellement, nous définissons la vue de supervision (*EFM-view*) comme suit.

DÉFINITION 3 (Vue de supervision (EFM-View)). — *Soit $\mathcal{VS}_i$ la vue de supervision du participant P_i et $\mathcal{V}_i$ sa vue locale. $\mathcal{VS}_i = \cup_{j \in Sub(P_i)} \mathcal{VS}_j \cup \mathcal{V}_i$*

4. Algorithmes de configuration et d'échange de notifications

Pour permettre la mise en œuvre de notre approche d'échange de notification entre partenaires, nous commençons par distinguer deux phases: la phase de configuration et la phase d'exécution. La phase de configuration consiste à calculer le *super-partenaire* et les *sous-partenaires* de chaque participant et la définition des notifications nécessaires pour chaque modèle de chorégraphie (i.e. quelles sont les interactions qui doivent être notifiées et à quel partenaire). L'*algorithme* 1 montre un pseudo-code de l'algorithme de configuration.

Algorithme 1: Algorithme d'échange de notification (Phase de configuration)

```
1 Require: - C: un modèle de choreographie (une vue globale)
2 - P: un ensemble de participants
3 for (each P_i in P) do
      /* Identification of the Super of P_i                    */
4     I_i0 ← GetFirstInteraction(P_i)
5     Super_i ← GetSource(I_i0)
6     SubPartners(Super_i) ← SubPartners(Super_i) ∪ {P_i}
      /* Définition de l'ensemble des notifications
      nécessaires                                              */
7     V_i ← ConstuctNotificationSet(P_i)
```

Soit C est une chorégraphie et $\mathcal{P}$ l'ensemble des participants. Pour chacun des participants P_i, nous cherchons la première interaction dans sa vue locale et nous regardons le champ «émetteur» dans le message associé. Ce dernier est considéré comme responsable de l'introduction de P_i dans la collaboration (i.e. son *super-partenaire*). Ensuite, nous définissons, à partir du modèle défini de la chorégraphie, l'ensemble des notifications nécessaires (i.e. celles générées pendant la phase d'exécution).

Algorithme 2: Algorithme d'échange de notifications (Phase d'exécution)

```
1  Require: - S: Le super partenaire du participant en question
2  - V: La vue de supervision EFM-view
3  for (each event occurrence e_i ) do
4      if (e_i.type ≠ Exception and CheckConformance(e_i)) then
5          UpdateMonitorState(e_i)
6          if (e_i.msrc ≠ Super and e_i.mdst ≠ Super) then  /* Generate a new
           notification if e_i is not coming from/going to the
           Super */
7              if (e_i.type = message exchange) then
8                  n ← GenerateNotification(e_i)
9                  SendNotification(n, Super)
10             else
                   /* e_i.type = notification exchange                  */
11                 ForwardNotification(e_i, Super)
12     else
13         AlertInternalMonitor()
14         ReportExceptionTo(Super, Sub(P_i))
```

L'*algorithme* 2 décrit le processus d'échange de notifications qui est exécuté au niveau de chaque participant lors de la phase d'exécution. Cet algorithme montre comment et quand générer, envoyer et transférer les notifications pour chacun des EFMs. Chaque EFM traite trois types d'événements: (i) les événements liés aux messages échangés avec d'autres partenaires (messages de chorégraphie), (ii) les événements liés au transfert de notifications vers le sommet de l'hiérarchie (i.e. les événements liés aux messages échangés entre des sous-partenaires), et (iii) les événements indiquant des exceptions. Selon le type d'événement, l'EFM se comporte différemment:

– si l'événement est un échange de message, l'EFM vérifie la conformité avec sa vue de supervision ($EFM - view$). Cela consiste à vérifier si le message reçu ou envoyé est conforme à celui qui est attendu en se basant sur les contraintes spécifiées dans le modèle de chorégraphie. Cela permet, entre autres, d'éviter l'envoi de notifications inappropriées ou la duplication des notifications. Si le message est conforme, l'EFM met à jour son statut (noeud actuel dans le graphe de vue de supervision), génère une nouvelle notification indiquant l'occurrence du message et l'envoie à son *super-partenaire*. Il est inutile de notifier le *super-partenaire* si ce dernier figure dans le champ émetteur ou receveur du message car il est déjà au courant de son occurrence.

– si l'événement est une réception de notification (i.e. un des *sous-partenaires* nous a envoyé une notification sur une occurrence de message), l'EFM vérifie si c'est conforme à sa vue locale et met à jour son statut. Ensuite, il transfère la notification à son *super-partenaire*. Dans ce cas aussi, nous optimisons le nombre de notifications envoyées en éliminant celles qui ne sont pas nécessaires (e.g. quand le *super-partenaire* est déjà au courant de l'évènement). L'EFM ne transfère donc pas les évènements ayant son *super-partenaire* comme émetteur ou receveur du message notifié.

– si l'événement est une exception générée par un autre partenaire, l'EFM traite localement cette exception et la transmet à son *super-partenaire* et à tous ses *sous-partenaires*. Les autres font de même afin que l'exception soit propagée.

Dans les deux premiers cas, si l'événement n'est pas conforme à celui attendu, une exception est générée, le moniteur interne est alerté. Le *super-partenaire* et tous les *sous-partenaires* sont ensuite informés. Il est à noter que les *super-partenaires* peuvent déceler l'exception automatiquement sans avoir à être notifiés par leurs *sous-partenaires* car ils seront bloqués après un certain temps. Une telle propagation rapide des exceptions dans tout l'arbre permettra d'accélérer la détection des situations de blocage.

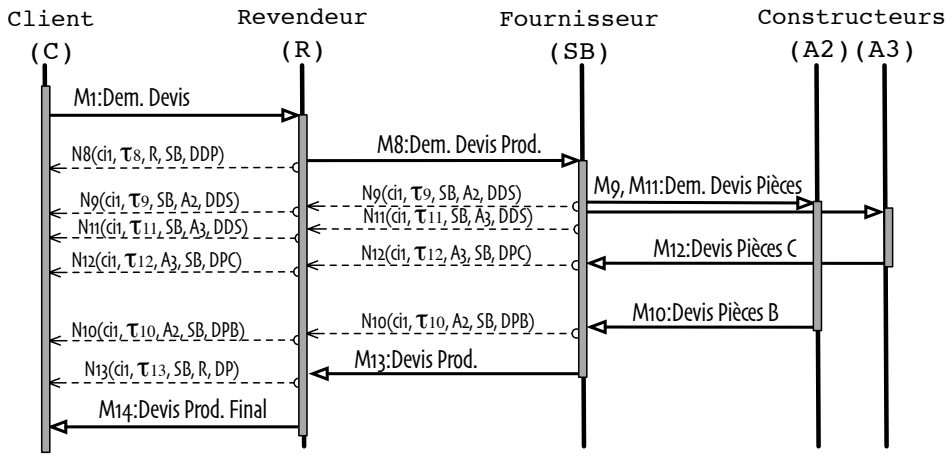


Figure 5. Exemple d'une exécution correcte (Diagramme de séquence)

La *figure* 5 montre, à l'aide d'un diagramme de séquence, une exécution réussie. Les messages de la chorégraphie sont modélisés par des flèches continues et les notifications échangées par des flèches en pointillé. $ci1$ est l'identificateur de l'instance. τ_i représente le $timestamp$ de l'occurrence du message M_i (temps de réception ou d'envoi). En fait, l'attribut $timestamp$ est nécessaire pour la vérification de l'ordre des messages. Les trois derniers attributs de la notification correspondent à l'émetteur, receveur et type du message associé. La chorégraphie est correcte car les différents messages et notifications sont conformes aux attentes de chacun des participants. Par exemple, le revendeur R sait que les échanges entre le fournisseur et les constructeurs se sont déroulés conformément à ce qui était prévu grâce aux notifications reçues de SB (N_9 à N_{12}). De plus, il reçoit le message attendu de SB (M_{13}).

5. Mise en œuvre de la supervision décentralisée

5.1. Supervision événementielle dans un environnement CEP

À partir du schéma d'interaction obtenu à partir de l'algorithme 2, nous pouvons générer un certain nombre de requêtes qui vont permettre de vérifier que le comportement réel de la chorégraphie adhère aux contraintes métier imposées. Cette approche permet de vérifier ensuite la conformité des séquences d'interaction par rapport au modèle de la chorégraphie, et ce, à chaque échange de message entre les partenaires.

La vérification se fait au niveau de chaque participant par rapport à sa vue de supervision *EFM-view*. Par rapport à d'autres approches existantes (Weidlich *et al.*, 2011), notre objectif est de réduire au maximum le nombre de règles sur le séquencement des interactions en ne fixant que des contraintes entre les interactions voisines (et par conséquent le nombre de messages).

Pour chaque exécution d'une instance de chorégraphie telle que celle décrite dans la figure 5, nous enrichissons les événements provenant du flux d'événements de chaque partenaire en ajoutant un attribut « ascendancy » qui contient la liste des blocs ascendants (i.e. contenant l'interaction associée) dans l'arbre dérivé du modèle de chorégraphie. Pour repérer les violations de conformité au cours de l'exécution de la chorégraphie, les contraintes de séquencement des messages sont alors transformées en requêtes sur les événements. Le flux d'événements générés est vérifié par rapport à ces requêtes, et ce, à chaque arrivée d'un nouvel événement au cours de l'exécution. Nous verrons dans la section suivante comment nous minimisons le nombre d'alertes en regroupant les violations détectées suivant leurs causes principales. En d'autres termes, toutes les violations ayant la même cause principale sont agrégées en une seule violation contenant la date, le type et les différentes répercussions. Les violations agrégées peuvent ensuite être affichées dans les tableaux de bord de supervision.

Dans le domaine des processus métier, une approche se basant sur le concept appelé profil de causalité comportementale (*causal behavioral profiles*) montre comment résoudre le problème en proposant de générer une règle entre chaque couple d'activités (Weidlich *et al.*, 2011 ; 2010). Cette approche fournit une abstraction du comportement défini par un modèle de processus en capturant les caractéristiques comportementales par des relations binaires. Autrement dit, une relation est fixée pour chaque couple d'activités indiquant si les deux activités doivent s'exécuter dans un ordre strict, exclusivement l'une de l'autre ou sans aucune contrainte entre elles. Chaque relation d'ordre entre deux activités est transformée en une requête CEP (*Complex Event Processing*) qui permet de détecter si l'inverse (appelé anti-patron) de cette relation se produit. Ainsi, une relation d'exclusion entre deux activités est violée si et seulement si une requête détecte l'exécution de deux activités ensemble dans la même instance.

Dans une chorégraphie, une interaction consiste en un message d'un certain type entre deux entités, la source et la destination. On recense par exemple dans le modèle présenté dans la *figure* 6, les interactions (I_1) et (I_3) (qui respectent un ordre strict), les interactions (I_4) et (I_7) (qui sont exclusives l'une à l'autre), et les interactions (I_4) et (I_8) (sans séquencement particulier).

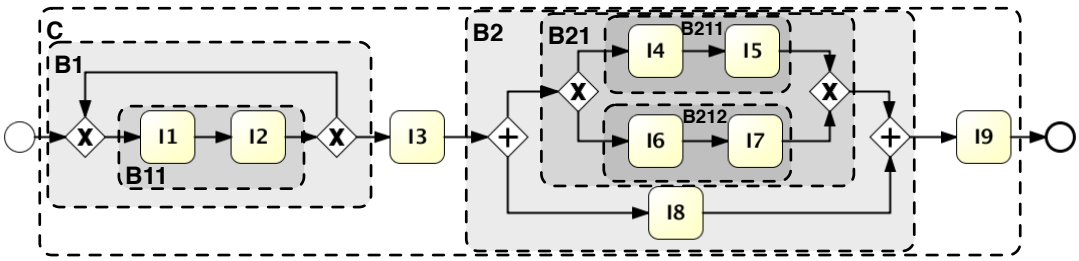


Figure 6. Décomposition en fragments

Dans l'approche de Wiedlich, le nombre de relations (i.e. une relation pour toute combinaison de couple de d'interaction), et donc de requêtes, est en $O(n^2)$ pour toute chorégraphie ayant n interactions (précisément $n(n+1)/2$ si on ne prend pas les règles qu'on peut déduire symétriquement). Nous pouvons remarquer qu'il y a beaucoup de relations non nécessaires (i.e. qui peuvent être déduites indirectement). Par exemple, puisque «(I_1) précède (I_3)» et «(I_3) précède (I_4)», nous pouvons directement déduire que «(I_1) précède (I_4)». Nous raisonnons de manière identique pour (I_1) par rapport à (I_5),..., (I_9). Cela est imputable du fait que certaines contraintes de séquencement sont transitives (e.g. les relations «précède» et «suit»).

Lorsque l'on génère les requêtes CEP, il existe un risque important d'obtenir un nombre non négligeable de requêtes qui se chevauchent. De plus, lorsqu'une interaction est effectuée à un stade inattendu (i.e. violation), les requêtes générées peuvent entraîner de multiples alertes redondantes. En effet, lorsqu'il y a violation, nous pouvons avoir plusieurs requêtes qui lèvent la même exception et plusieurs alertes seront générées et envoyées pour la même violation (de par la transitivité de certaines relations). Par exemple, si un message M_9 correspondant à l'interaction I_9 est envoyé avant tous les autres messages, nous aurons 8 violations (I_9 par rapport I_1, I_9 par rapport I_2, etc.) alors qu'il s'agit d'une seule violation avec 7 répercussions.

Ce problème a été traité a posteriori dans (Weidlich *et al.*, 2011). Des requêtes supplémentaires sont ajoutées afin d'identifier la cause principale de tout un ensemble de violations. Cependant, même avec cette amélioration, la détermination de l'ensemble de violations en relation reste toujours une phase délicate et varie pour chaque patron de séquencement. Enfin, le problème des boucles n'est pas traité.

Nous prenons en compte ces problèmes dans notre approche. Au lieu de fixer une contrainte entre chaque couple d'interaction, notre approche consiste à fixer des contraintes uniquement entre les interactions de voisinage. Pour ce faire, une fragmentation structurelle du modèle de chorégraphie est nécessaire.

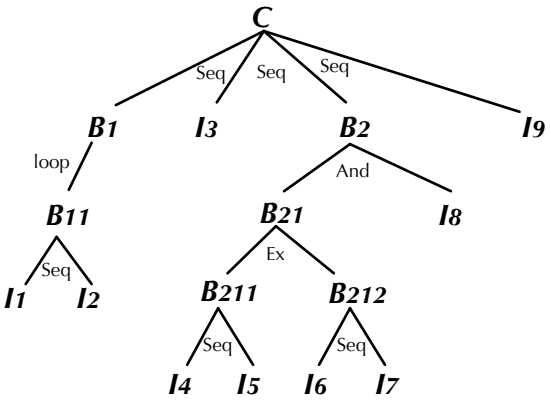
5.2. Fragmentation structurelle d'une chorégraphie et événements de blocs

En nous inspirant de la décomposition des processus métier de l'arbre raffiné de structure de processus (R-PST : *Refined Process Structure Tree*) (Polyvyanyy *et al.*, 2010), nous décomposons le modèle de chorégraphie . Cette technique consiste à analyser et à découvrir la structure du graphe de flux de contrôle d'un processus donné. Elle permet une décomposition hiérarchique d'un modèle de processus en un ensemble de blocs *canoniques* avec un seul flux d'entrée et un seul flux de sortie (SESE : *Single Entry/ Single Exit*). Un bloc *canonique* est défini comme suit (Vanhatalo *et al.*, 2007).

DÉFINITION 4 (Bloc canonique). — *Un bloc B est dit* non-canonique *si et seulement s'il existe trois blocs X, Y et Z avec ces trois conditions réunies: a) X et Y sont en séquence, b) $B = X \cup Y$, c) B et Z sont en séquence.*
Sinon, B est dit canonique.

Il a été prouvé que la décomposition R-PST est unique, modulaire et peut être calculée en temps linéaire (Polyvyanyy *et al.*, 2010). En effet, les blocs *canoniques* ne se chevauchent jamais. Prenant deux blocs, soit un bloc est entièrement contenu dans l'autre, soit les deux blocs sont totalement disjoints. La *figure* 6 illustre le résultat de la décomposition en une hiérarchie de blocs *canoniques* SESE de notre exemple.

La décomposition peut alors être représentée sous forme d'un arbre (appelé arbre de structure de chorégraphie (CST: *Choreography Structure Tree*)). La racine de l'arbre représente le plus grand bloc qui contient l'ensemble du graphe d'interactions. Les blocs descendants d'une séquence sont classés de gauche à droite (i.e. de l'entrée vers la sortie). La *figure* 7 montre l'arbre CST généré de notre exemple. Nous avons ainsi explicitement établi les relations structurelles de voisinage entre les blocs.



$$I_1 \rightarrow generate(E_{id}, C_{id}, I_1, < \mathbf{B_{11}}, \mathbf{B_1} >, TS)$$
$$I_2 \rightarrow generate(E_{id}, C_{id}, I_2, < \mathbf{B_{11}}, \mathbf{B_1} >, TS)$$
$$I_3 \rightarrow generate(E_{id}, C_{id}, I_3, <>, TS)$$
$$I_4 \rightarrow generate(E_{id}, C_{id}, I_4, < \mathbf{B_{211}}, \mathbf{B_{21}}, \mathbf{B_2} >, TS)$$
$$I_5 \rightarrow generate(E_{id}, C_{id}, I_5, < \mathbf{B_{211}}, \mathbf{B_{21}}, \mathbf{B_2} >, TS)$$
$$I_6 \rightarrow generate(E_{id}, C_{id}, I_6, < \mathbf{B_{212}}, \mathbf{B_{21}}, \mathbf{B_2} >, TS)$$
$$I_7 \rightarrow generate(E_{id}, C_{id}, I_7, < \mathbf{B_{212}}, \mathbf{B_{21}}, \mathbf{B_2} >, TS)$$
$$I_8 \rightarrow generate(E_{id}, C_{id}, I_8, < \mathbf{B_2} >, TS)$$
$$I_9 \rightarrow generate(E_{id}, C_{id}, I_9, <>, TS)$$

Figure 7. Enrichissement des événements

5.3. Enrichissement des événements de base et de haut niveau (END-events)

Après avoir généré l'arbre CST, nous pouvons alors enrichir chaque événement en ajoutant un nouveau champ appelé *ascendancy* contenant la liste de tous les blocs supérieurs de l'interaction observée. Par exemple, l'événement lié à l'interaction (I_4) dans notre exemple est marqué par une séquence de blocs $< B_{211}$,B_{21},$B_2 >$, car il appartient respectivement aux blocs B_{211}, B_{21} et B_2.

Cette étape consiste, à chaque réception d'un nouvel événement, à retrouver dans l'arbre CST la liste des blocs qui contiennent l'interaction correspondante à l'événement arrivé et à attacher le résultat trouvé dans le champ <ascendancy>. La *figure* 7 illustre comment enrichir les événements à partir de l'arbre CST.

Après avoir enrichi les événements de base avec leurs blocs parents, les contraintes de séquencement peuvent désormais être appliquées non seulement sur les événements mais aussi sur les blocs. Nous pouvons ainsi fixer des relations binaires entre deux événements, entre deux blocs, ou encore entre un événement et un bloc. En agissant ainsi, nous fixons uniquement des relations entre les voisins directs et faisons donc décroitre donc le nombre de requêtes de façon exponentielle du fait de la structure arborescente des blocs. Ceci représente un des avantages majeurs de notre approche.

Pour permettre d'établir des relations entre blocs, nous avons besoin de générer des événements de plus haut niveau qui symbolisent l'exécution de tout un bloc. Pour ce faire, nous avons choisi de définir une nouvelle structure d'événement que nous appelons *évènement de fin de bloc* (*END-events*). Nous notons ainsi $End(B)$ l'évènement de fin d'exécution du bloc B.

DÉFINITION 5 (Événements de haut niveau). — *Dans un CEP, un événement est dit de haut niveau s'il représente une abstraction d'autres événements, appelés ses membres (Luckham, 2002). Dans notre cas, les membres d'un événement $End(B)$ sont tous les événements liés aux interactions et aux blocs contenus dans le bloc B.*

Exemple: Dans la *figure* 6, les événements associés à I_1, I_2 et $End(B_{11})$ sont les membres de l'événement de plus haut niveau $End(B_1)$.

5.3.1. Génération automatique pour une séquence

à titre d'exemple, nous montrons comment générer les règles permettant d'exprimer une séquence entre deux blocs. On pourra trouver dans (Baouab, 2013) tous les détails pour les autres règles. On spécifie d'abord les contraintes de séquencement sous forme de relations binaires. Ensuite, on définit les règles de voisinage sur les blocs ainsi que les règles de base pour la génération pour chaque type de patron.

La relation « séquence » permet de définir un ordre strict entre deux événements, deux blocs ou encore un bloc et un événement. La table 1 montre les différentes combinaisons possibles.

$Seq(I_i, I_j)$	Pour deux événements I_i et I_j, I_j ne se produit jamais avant I_i.
$Seq(B_i, B_j)$	Pour deux événements I_i et I_j, respectivement marqués avec B_i et B_j, I_j ne se produit jamais avant la fin de B_i ($End(B_i)$).
$Seq(I_i, B_j)$	Pour deux événements I_i et I_j, avec I_j marqué par B_j, I_j ne se produit jamais avant I_i.
$Seq(B_i, I_j)$	Pour deux événements I_i et I_j, avec I_i marqué par B_i, I_j ne se produit jamais avant la fin de B_i ($End(B_i)$).

TABLE *1. La relation « séquence »*

5.3.2. Règles de génération par patron

Selon les patrons de séquencement dans un modèle de chorégraphie (i.e. séquence, parallèle, exclusion et itération), nous pouvons maintenant définir les règles de génération automatique des relations binaires ainsi que les événements de fin de bloc. La *figure* 8 illustre les règles pour chaque type de patron.

Pour un bloc de type *Séquence*, lorsque nous avons un bloc B qui contient n blocs d'interaction en séquence $B_1, ..., B_n$, nous générons uniquement $n-1$ relations binaires entre chaque deux blocs consécutifs, c'est-à-dire $Seq(B_i, B_{i+1}), i \in \{1..n-1\}$.

L'achèvement du dernier bloc dans la liste provoque la génération de l'événement de fin du bloc B, c'est-à-dire $End(B_n) => Generate(End(B))$.

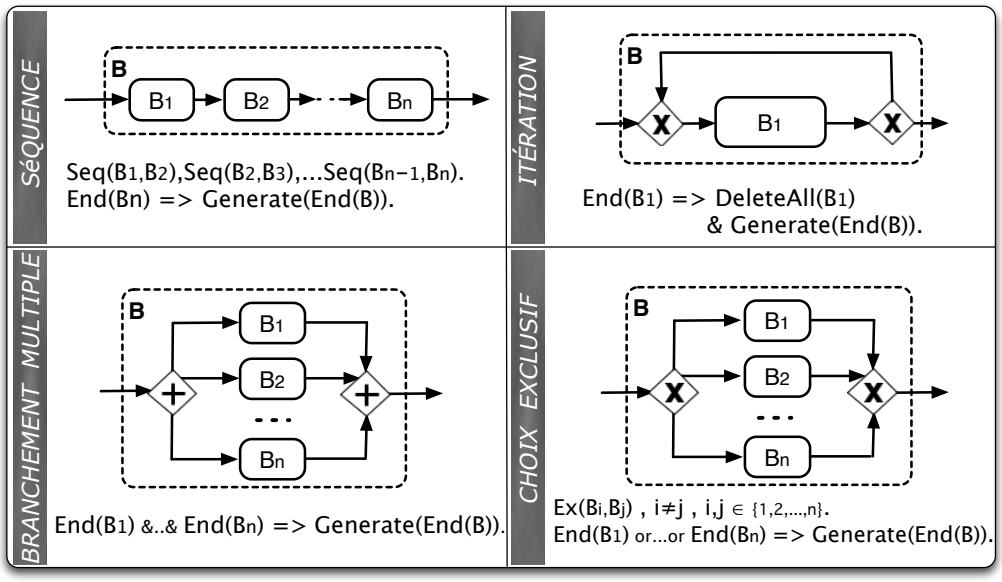


Figure 8. Règles par patron

On traite suivant le même principe les blocs *Parallèle*, *Choix exclusif* et *Itération*.

5.4. Évaluation du nombre de requêtes générées

En appliquant les quatre règles définies dans le paragraphe précédent, un ensemble de relations optimisé et spécifique à chaque modèle de chorégraphie peut être généré automatiquement en explorant l'arbre CST niveau par niveau. Dans chaque niveau de l'arbre, les relations binaires entre les noeuds voisins sont générées selon l'annotation du noeud parent qui définit le type de patron de séquencement. Ensuite, les règles de génération des événements de fin de bloc sont à leur tour générées.

La table 2 affiche l'ensemble des relations et des règles générées à partir de notre exemple de modèle de chorégraphie (revoir l'arbre CST de la *figure* 7). Nous classifions les règles par leur niveau (i.e. profondeur dans l'arbre CST). Nous séparons en deux colonnes les relations binaires et les règles de génération.

Niv.	Relations	Règles de génération
1	$Seq(B_1, I_3)$ $Seq(I_3, B_2)$ $Seq(B_2, I_9)$	$I_9 \Rightarrow generate(End(C))$
2		$End(B_{11}) \Rightarrow deleteAll(B_{11})$ & $generate(End(B_1))$ $End(B_{21})$ & $I_8 \Rightarrow generate(End(B_2))$
3	$Seq(I_1, I_2)$ $Ex(B_{211}, B_{212})$	$I_2 \Rightarrow generate(End(B_{11}))$ $End(B_{211})$ or $End(B_{212}) \Rightarrow generate(End(B_{21}))$
4	$Seq(I_4, I_5)$ $Seq(I_6, I_7)$	$I_5 \Rightarrow generate(End(B_{211}))$ $I_7 \Rightarrow generate(End(B_{212}))$

TABLE *2. Relations et règles de fin de bloc générées*

Remarque: Nous ajoutons la relation $Ex(I_i, I_i)$ pour imposer le fait que chaque interaction ne peut pas se produire plusieurs fois. Cette relation n'affecte pas les événements associés à des interactions d'un bloc de boucle car ces événements sont supprimés à la fin de chaque itération.

Nous pouvons remarquer une différence importante en terme de nombre de règles et relations générées par rapport à l'approche classique (Weidlich *et al.*, 2011). En effet, rien que sur cet exemple, notre approche permet de réduire le nombre de relations binaires à 7 (voir 2ème colonne de la *table* 2). En plus de ces 7 relations, nous avons 7 règles de génération d'événements de haut niveau (voir 3ème colonne du tableau) ce qui nous ramène à un total de 15 (au lieu de $9^2 = 81$ relations avec l'approche mentionnée ci-dessus.

Bien entendu, le gain apporté par notre approche dépend de la topologie de l'arbre CST (i.e. le nombre moyen d'interactions par bloc, les types de patrons de séquencement et leurs combinaisons). Dans le cas de N interactions en séquence, nous devons générer $N-1$ relations binaires entre chaque deux blocs consécutifs. à la détection de l'événement associé à la dernière interaction I_N, l'événement de fin de bloc $End(B)$ sera généré. Ceci ramène le nombre total de règles à N au lieu de N^2 avec l'approche classique (voir *figure* 9).

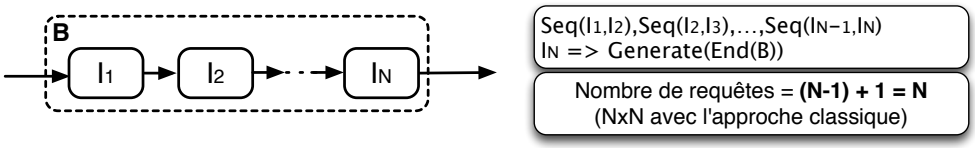


Figure 9. Cas 1 : séquence de N interactions

Si on analyse le cas de deux blocs B_1 et B_2 en parallèle contenant respectivement N et M interactions en séquence, nous générons $N-1$ relations binaires de séquence et un événement de fin de bloc pour chacun des deux blocs. La génération des deux événements de fin de bloc va permettre au bloc suivant de s'exécuter. La *figure* 10 illustre l'exemple avec l'ensemble des règles générées.

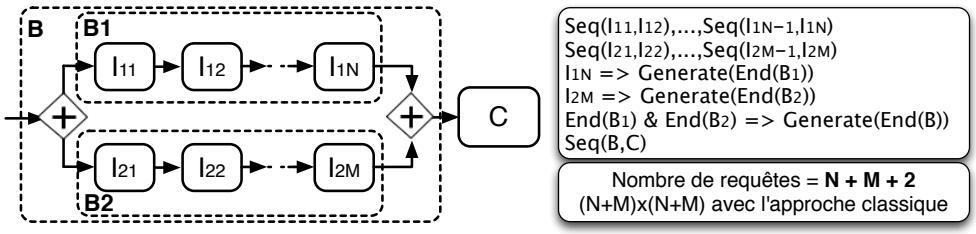


Figure 10. Cas 2: blocs en parallèle

Ces deux exemples illustrent la façon dont notre approche permet de réduire le nombre de règles nécessaires pour la supervision et la détection des violations. Cette optimisation va permettre non seulement une meilleure lisibilité, mais aussi une réactivité améliorée vu que le processeur de règles mettra moins de temps pour le traitement des requêtes. De plus, nous proposons une solution au traitement des itérations

6. Détection des violations

Afin de détecter les violations, nous allons transformer les relations d'ordre ainsi que les règles de génération de fin de bloc sous forme de requêtes CEP. La transformation se fait de façon automatique, et ce, règle par règle. Sachant que les requêtes générées vont être exécutées sur le flux des évènements reçus par le processeur CEP, les résultats de ces requêtes doivent correspondre à des violations détectées. Pour ce faire, il faut inverser les relations générées avant de les transformer en requêtes. En effet, chaque requête CEP générée doit correspondre à une négation d'une relation d'ordre. Par exemple, la relation $Seq(B_1, B_2)$ est violée, si le processeur CEP détecte un événement marqué par B_2 qui est suivi par un autre événement marqué par B_1 ou simplement suivi par l'événement de fin de bloc $End(B_1)$. Cependant, la relation $Ex(B_1, B_2)$ est violée lorsqu'il détecte deux événements, respectivement marqués avec B_1 et B_2, ayant le même identifiant d'instance.

```
 // Détecter les violations de la relation Seq(I1,I2) :
"@Name('Seq I1 I2') select * from pattern "
    + "[ (e2=MsgEvent(e2.iid=2) and not e1=MsgEvent(e1.iid=1))] where e1.cid=e2.cid" ,

// Détecter les violations de la relation Ex(B211,B212) :
"@Name('Ex B211 B212') select * from pattern "
    + "[ e1=MsgEvent(e1.ascendancy like '%B211,%') and "
    + "e2=MsgEvent(e2.ascendancy like '%B212,%')] where e1.cid=e2.cid",

// Détecter les violations de la relation Seq(B1,I3) :
"@Name('Seq B1 I3') select * from pattern "
    + "[ e3=MsgEvent(e3.iid=3) and not b1=MsgEvent(b1.endOf like 'B1')] "
    + "where e3.cid=b1.cid",
/*...*/
```

Figure 11. Formulation de requêtes CEP avec le langage Esper

La *Figure* 11 montre trois requêtes de notre exemple de motivation formulées avec le langage Esper (un langage de traitement des événements CEP (EsperTech, 2011)).

La première requête correspond à la négation de la relation $Seq(I_1, I_2)$. Elle permet de détecter si un événement $e1$ de type I_1 n'est pas précédé par un autre événement $e2$ de type I_2 appartenant tous les deux à la même instance de chorégraphie ($e1.Cid = e2.Cid$).

La seconde requête correspond à la négation de la relation $Ex(B_{211}, B_{212})$. Elle permet de détecter si deux événements $e1$ et $e2$ marqués respectivement par B_{211} et B_{212} appartiennent à la même instance de chorégraphie ($e1.Cid = e2.Cid$).

La troisième requête montre le cas d'une séquence entre un bloc et un message. La requête permet de détecter si la relation $Seq(B_1, I_3)$ a été violée (tout événement de type I_3 qui n'est pas précédé par l'événement de fin de bloc $End(B_1)$).

Grâce à la définition de ces anti-patrons, nous sommes en mesure de détecter plusieurs situations. Par exemple, nous pouvons détecter une violation d'ordre. Cette violation se produit lorsque l'ordre des messages n'est pas cohérent avec le comportement défini. Considérons, à titre d'exemple, notre exemple de chorégraphie avec la séquence des événements $< I_1, I_2, I_4, I_8, I_3, I_5, I_9 >$ reçus par le processeur CEP. En se référant aux règles relations générées dans la *table* 2, nous pouvons remarquer

que la relation $Seq(I_3, B_2)$ est violée par chacun des deux événements I_4 et I_8, qui sont étiquetés avec B_2, car ils ont eu lieu avant l'événement I_3. La *figure* 12 schématise les violations sur le modèle.

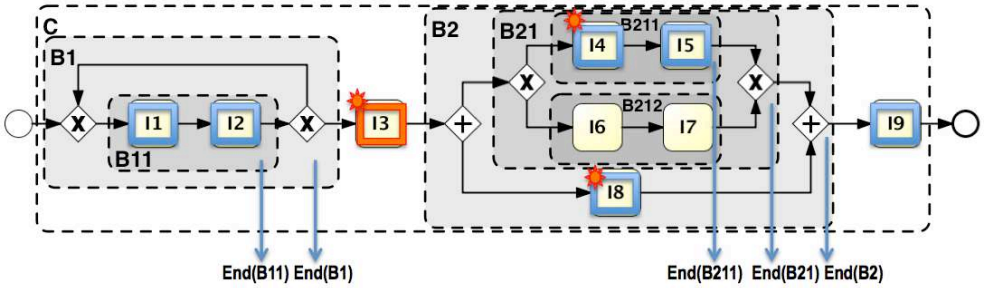


Figure 12. Violation d'ordre de messages: $< I_1, I_2, I_4, I_8, I_3, I_5, I_9 >$

Nous pouvons également détecter un message manquant dans une séquence. Cette violation est détectée en cas d'une non occurrence d'un message. Les violations de ce type ne peuvent être matérialisées qu'à la fin d'exécution du plus petit bloc contenant l'interaction en question. En fait, lorsque l'événement de fin de ce bloc n'est pas généré, la séquence suivante est nécessairement violée.

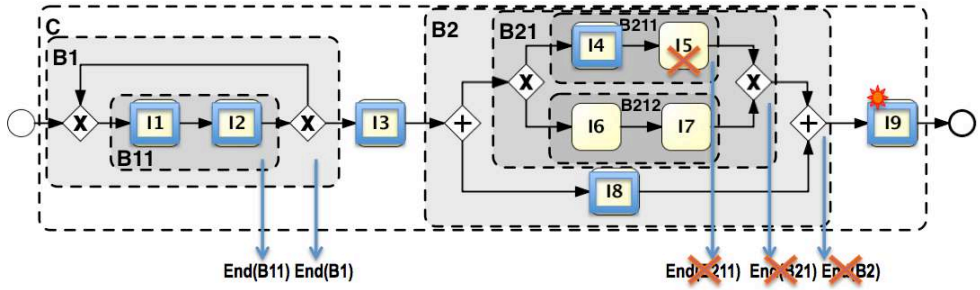


Figure 13. Message manquant: $< I_1, I_2, I_3, I_8, I_4, I_9 >$

Par exemple, soit $< I_1, I_2, I_3, I_8, I_4, I_9 >$ la séquence des événements générés. Comme nous pouvons le remarquer, l'événement associé à I_5 est manquant dans la séquence. Dans ce cas, l'événement de fin de bloc $End(B_{211})$ ne sera pas généré (règle de génération: $I_5 \Rightarrow generate(End(B_{211}))$). Par conséquent, les événements de fin de blocs $End(B_{21})$ et $End(B_2)$ ne seront non plus générés. Comme l'événement I_9 a eu lieu avant l'événement de fin de bloc $End(B_2)$, la relation $Seq(B_2, I_9)$ est alors violée. La *figure* 13 schématise les violations sur le modèle.

7. Agrégation des violations

Lorsque l'on détecte une ou plusieurs violations, il devient intéressant de traiter ces violations afin de fournir une explication qui soit la plus claire possible sur l'origine du problème. Nous présentons maintenant une méthode qui permet la collecte, le tri et l'agrégation des violations détectées au cours de l'exécution. Le but est de minimiser le nombre d'alertes et d'ajouter une nouvelle information ayant plus de valeur sémantique et permettant d'offrir à l'administrateur une meilleure visibilité sur l'origine et les relations qui existent entre les violations détectées.

Lors de l'exécution d'une instance de chorégraphie, plusieurs violations peuvent survenir. Dans notre approche, une violation est détectée à chaque fois qu'une des relations binaires n'est pas satisfaite. Une relation est violée quand deux interactions ne se produisent pas dans l'ordre (cas de la relation Seq) ou bien quand elles ne sont pas exclusives l'une à l'autre (cas de la relation Ex). Cependant, il arrive qu'une même interaction soit à l'origine de deux ou plusieurs violations. Ainsi, des violations multiples peuvent être causées par un même message. Ajoutons à cela le fait qu'une même relation peut être violée plusieurs fois par différents messages (e.g. la violation d'une relation d'exclusion entre deux blocs A et B peut se produire plusieurs fois dans la même instance lorsque plusieurs interactions marquées avec le bloc A se produisent avec d'autres marquées avec le bloc B).

Considérons notre modèle de chorégraphie présenté précédemment. On suppose la trace d'événements $< I_1, I_2, I_5, I_3, I_4, I_6, I_9, I_7 >$. L'ensemble des violations détectées est représenté dans la *figure* 14. Nous remarquons ici que la relation $Ex(B_{211}, B_{212})$ a été violée quatre fois par différentes combinaisons de couples d'interactions (I_6 avec I_5, I_6 avec I_4, I_7 avec I_5 et enfin I_7 avec I_4).

Nous pouvons également remarquer que certaines interactions causent la violation de plus d'une relation. Par exemple, l'interaction I_5 a causé la violation de la relation $Seq(I_3, B_2)$ puisqu'elle est marquée avec le bloc B_2 et s'est produite avant l'interaction I_3. Cette même interaction entraine aussi la violation de la relation $Seq(I_4, I_5)$ en se produisant avant l'interaction I_4.

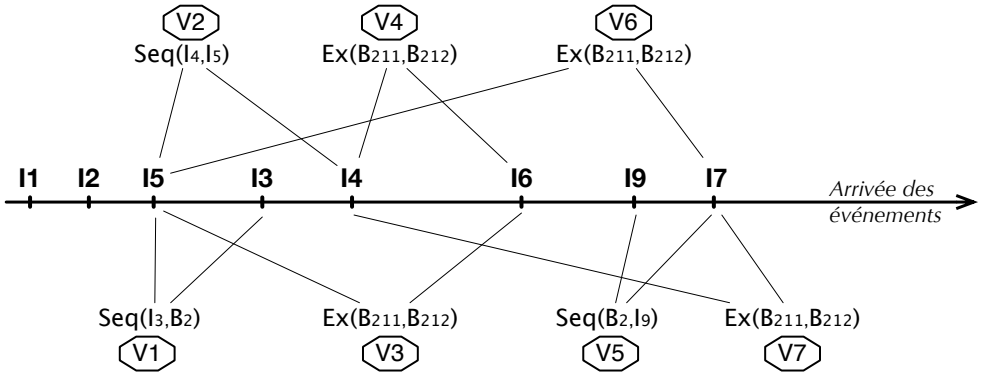


Figure 14. Exemple de violations enregistrées dans une instance de chorégraphie

Afin de faciliter la tâche de l'administrateur qui va recueillir toutes les violations et/ou adopter une compensation adéquate, nous avons décidé d'identifier la cause principale de toute violation et de n'en remonter que cette cause principale.

Afin de permettre l'agrégation des violations, nous commençons par définir la structure d'une violation dite « agrégée ». Dans cette structure, on trouve l'identité de l'événement, l'identité de l'instance de la chorégraphie, l'ensemble des relations violées, l'identifiant de l'interaction qui a provoqué la violation Iid_c, l'ensemble Iid_r des identifiants des interactions associées aux messages avec qui Iid_c est en conflit, et une estampille.

Après avoir défini la structure d'une violation agrégée, il faut définir une fonction qui permet l'agrégation de plusieurs violations atomiques en une seule agrégée. Cette fonction est basée sur l'algorithme suivant.

Algorithme 3: Algorithme d'agrégation de violations

```
1  Require: - v_n: La dernière violation arrivée
2  - T: La table des violations atomiques
3  - T_agr: La table des violations agrégées
4  Insert(T, v_n)
5  original ← true
6  for (each v_i ∈ T_agr ) do
7      if (v_i.Iid_c = v_n.Iid_c) then /* Si la cause de la violation existe
       comme cause dans la table */
8          Aggregate(v_i, v_n)
9          original ← false
10         Break ;                          /* Sortir de la boucle */
11 if (original = true) then   /* Violation originale non trouvée dans
   la table */
12     SendAlert(v_n)
13     GenerateNewAgr(v_n)
14     SendLastaggregation()
15 End
```

L'*algorithme* 3 décrit comment traiter chaque violation nouvellement arrivée. Cet algorithme permet de mettre à jour la table de violations atomiques et celle des violations agrégées qui sont deux tables créées pour stocker temporairement les violations en cours. En effet, chaque nouvelle violation est d'abord stockée dans la table des violations atomiques $\mathcal{T}$ (ligne 4) en initialisant la variable booléenne $original$ à «true» (ligne 5). Ensuite, nous parcourons la table des violations agrégées (ligne 6) pour chercher si l'interaction qui a causé la violation en cours existe comme cause d'une autre violation. Si c'est le cas, nous fusionnons cette violation atomique avec la violation trouvée et la variable $original$ devient «false» afin d'indiquer que la violation n'est pas la cause initiale (lignes 7 à 10). Enfin, si l'interaction n'est pas trouvée comme cause, la violation atomique est envoyée au tableau de bord et une nouvelle entrée est ajoutée dans la table des violations agrégées (lignes 11 à 14).

Remarque: D'un point de vue technique, il est nécessaire de spécifier une fenêtre coulissante (*time window*) durant laquelle le processeur CEP doit rester en attente avant de générer un événement agrégé. Cette fenêtre permet de limiter le nombre d'événements considérés par une requête CEP comme expliqué dans (EsperTech, 2011).

Les deux tables 3 et 4 montrent respectivement la table des violations atomiques et celle des violations agrégées pour la séquence d'événements présentée dans la *figure* 14. Nous remarquons ici par exemple que l'arrivée de l'événement correspondant à I_5 a causé deux violations atomiques (les deux premières lignes de la table des violations

atomiques) qui sont ensuite regroupées en une seule violation agrégée (1ère ligne de la table des violations agrégées).

Vid	Relation violée (VR)	(Iid_c)	(Iid_r)
1	$Seq(I_3, B_2)$	I_5	I_3
2	$Seq(I_4, I_5)$	I_5	I_4
3	$Ex(B_{211}, B_{212})$	I_6	I_5
4	$Ex(B_{211}, B_{212})$	I_6	I_4
5	$Seq(B_2, I_9)$	I_7	I_9
6	$Ex(B_{211}, B_{212})$	I_7	I_5
7	$Ex(B_{211}, B_{212})$	I_7	I_4

TABLE *3. Violations atomiques*

Vid	Relations violées ($VR[]$)	(Iid_c)	($Iid_r[]$)
1	$Seq(I_3, B_2)$, $Seq(I_4, I_5)$	I_5	I_3, I_4
2	$Ex(B_{211}, B_{212})$	I_6	I_5, I_4
3	$Seq(B_2, I_9)$, $Ex(B_{211}, B_{212})$	I_7	I_9, I_5, I_4

TABLE *4. Violations agrégées*

8. Travaux connexes

La supervision des compositions de services a fait l'objet de plusieurs travaux de recherche. (Subramanian *et al.*, 2008) ont présenté une approche pour améliorer les moteurs BPEL en proposant un nouveau moteur appelé "SelfHealBPEL" qui met en œuvre des fonctionnalités supplémentaires pour la détection des violations et le traitement des exceptions. Cette approche peut avoir un impact négatif sur la performance et l'agilité du fait qu'il n'y a pas de séparation entre la logique de supervision et le moteur BPEL. (Barbon *et al.*, 2006) ont proposé une architecture qui sépare la logique métier d'un service de la partie de supervision. Cependant, leur approche centralisée cible les orchestrations BPEL et ne traitent pas le problème des chorégraphies dans un cadre inter-organisationnel. (Ardissono *et al.*, 2008) ont présenté un *framework* permettant le suivi de l'exécution d'une chorégraphie afin d'assurer la détection précoce des conflits et la notification des participants concernés. L'approche consiste en un moniteur central qui est notifié par chaque participant à chaque fois qu'il envoie ou reçoit un message. Néanmoins, cette approche est centralisée et reste non adaptée aux chorégraphies inter-organisationnelles quand il est difficile de trouver une partie commune auquelle toutes les organisations font confiance (e.g. cas de plusieurs pays).

Dans le cas des processus décentralisés au sein de la même organisation (i.e. même cercle de confiance), (Chafle *et al.*, 2004) ont modélisé un moniteur implémenté comme un service Web. Ce moniteur maintient l'état de toutes les activités du service composite. (Halle, Villemaire, 2009) ont introduit le concept de superviseur *MBM* qui intercepte et traite les messages échangés. Chaque participant a un *MBM* local qui signe les messages sortants avec l'état actuel pour éviter les désynchronisations et offrir un protocole d'échange fiable.

Concernant les solutions basées sur les événements, la plupart des approches sont basées sur la technologie BAM (Business Activity Monitoring) (Dahanayake *et al.*, 2011). (Kikuchi *et al.*, 2007) ont utilisé un format d'audit commun qui permet le traitement et la corrélation d'événements entre plusieurs moteurs BPEL. (Wetzstein *et al.*, 2010) ont présenté une spécification d'événements complexes et utilisé un identificateur d'instance de chorégraphie (CIID) pour permettre la corrélation des événements inter-organisationnels (ce qui n'est pas pris en charge dans (Kikuchi *et al.*, 2007)). Contrairement à ces travaux, nous avons proposé un mécanisme automatisé et décentralisé pour l'échange de notifications entre les participants dans le but de superviser des chorégraphies franchissant les limites des zones de confiance.

En ce qui concerne les travaux relatifs à la vérification basée sur l'approche événementielle, les travaux les plus proches sont ceux de Weidlich (Weidlich *et al.*, 2011). Ils reposent sur une technique formelle pour générer, à partir d'un modèle de processus, des requêtes CEP qui, une fois exécutées, permettent de détecter les comportements anormaux au cours de l'exécution du processus. Ils utilisent le concept de profil de causalité comportemental «*Behavioral Profile*» défini dans (Weidlich *et al.*, 2010). Les auteurs proposent de générer une règle d'ordre entre chaque couple d'activités du processus. L'approche de supervision basée sur les profils comportementaux permet de générer une requête CEP pour chacune des règles binaires. Cela permet donc de détecter tout comportement qui n'est pas conforme au modèle prédéfini du processus.

La principale limite de l'approche est que l'ensemble de règles n'est pas optimal. En effet, il y a plusieurs règles qui se chevauchent et d'autres qui ne sont pas nécessaires vu qu'elles peuvent être inférées (i.e. déduites indirectement à partir d'autres règles). De plus, lorsqu'une de ces règles est violée, nous pouvons avoir toute une série de répercussions sur d'autres règles indirectement dépendantes à la violation d'origine. Ainsi, des requêtes supplémentaires doivent être ajoutées afin d'identifier la cause fondamentale de l'ensemble des infractions.

9. Conclusion et perspectives

Nous avons dans ce papier présenté un mécanisme d'échange de notifications entre les différents participants d'une chorégraphie. Nous avons montré comment une telle approche pourra permettre une supervision décentralisée de l'exécution d'une chorégraphie, et ce, dans le but d'offrir plus de maîtrise au niveau de chaque organisation participante et de réduire les délais et les coûts en cas d'occurrence d'exceptions. La supervision se fait de façon abstraite (i.e. uniquement sur les échanges de messages inter-organisationnels) sans pour autant dévoiler la logique métier de chaque entreprise. La propagation hiérarchique des données de supervision permet de limiter l'exposition des données et d'éviter une surcharge du réseau (notification sélective). En outre, notre approche est non intrusive dans le sens où elle se base sur l'écoute passive des messages de la chorégraphie sans aucune modification de la structure ou du contenu des messages (i.e. les processus ne sont pas altérés et ne sont pas conscients de la supervision et l'échange de notifications). Notre approche permet aussi d'of-

frir une traçabilité de l'exécution des processus. En effet, les informations recueillies pourraient être utilisées a posteriori pour mesurer la performance globale du processus et le suivi de la réalisation des objectifs de chaque organisation.

Nous avons également proposé une approche évènementielle permettant de générer un ensemble optimal de requêtes afin de permettre la surveillance de la conformité des séquences d'interaction. Au lieu de fixer une contrainte entre chaque couple d'interaction, notre approche consiste à ne fixer que des contraintes entre les interactions de voisinage. Pour ce faire, nous avons montré comment transformer le diagramme d'une chorégraphie en une hiérarchie de blocs canoniques et enrichir chaque évènement par ses blocs ascendants dans l'arbre de structure CST. Selon les patrons de séquencement (i.e. séquence, parallèle, exclusion et itération), nous avons défini des règles de génération automatique des relations binaires ainsi que les événements de fin de bloc. Ensuite, nous avons illustré comment dériver de façon automatique les relations et les règles trouvées sous formes de requêtes CEP. Nous avons aussi montré comment ces requêtes sont directement utilisées dans un environnement CEP en fournissant des directives de mise en œuvre.

Les travaux actuels concernent deux directions. La première est dédiée à la prise en compte de chorégraphies non structurées, et consiste à traiter les modèles dans lesquels il n'y a pas qu'un seul initiateur. La seconde vise à mettre en œuvre l'approche dans un contexte de supervision du contrôle d'accès au sein d'un réseau social d'entreprise afin de détecter des accès présentant des anomalies.

Bibliographie

Ardissono L., Furnari R., Goy A., Petrone G., Segnan M. (2008). An event-based model for the management of choreographed services. In *Proceedings of 9th international conference EC-Web*, LNCS 5183, Italy, September, 2008.

Baouab A. (2013). *Gouvernance et supervision décentralisée des chorégraphies inter-organisationnelles*. Thèse, Université de Lorraine.

Baouab A., Fdhila W., Perrin O., Godart C. (2012). Towards decentralized monitoring of supply chains. In *Proceedings of IEEE 19th ICWS*, USA, June, 2012.

Baouab A., Perrin O., Godart C. (2011). An event-driven approach for runtime verification of inter-organizational choreographies. In *Proceedings of IEEE International Conference on Services Computing, SCC*, USA, July, 2011.

Baouab A., Perrin O., Godart C. (2012). An optimized derivation of event queries to monitor choreography violations. In *Proceedings of 10th International Conference on Service-Oriented Computing - ICSOC*, China, November, 2012.

Baouab A., Perrin O., Godart C. (2013). Supervision Décentralisée des Chorégraphies de Services. In *31ème Congrès INFORSID.* Sorbonne, Paris, France, 2013.

Barbon F., Traverso P., Pistore M., Trainotti M. (2006). Run-time monitoring of instances and classes of web service compositions. In *Proceedings of IEEE ICWS*, USA, September, 2006.

Chafle G. B., Chandra S., Mann V., Nanda M. G. (2004). Decentralized orchestration of composite web services. In *Proceedings of the 13th international conference WWW*, USA, May, 2004.

Dahanayake A., Welke R. J., Cavalheiro G. (2011). Improving the understanding of bam technology for real-time decision support. *Int. Journal. Bus. Inf. Syst.*, vol. 7, p. 1–26.

EsperTech. (2011). *Esper - Complex Event Processing*. http://esper.codehaus.org

Francalanza A., Gauci A., Pace G. (2010). *Runtime monitoring of distributed systems*. Rapport technique. University of Malta. (WICT)

Halle S., Villemaire R. (2009). Flexible and reliable messaging using runtime monitoring. In *Workshops Proceedings of the 12th IEEE International EDOC Conference, EDOCw*, New Zealand, September, 2009.

Haq I. ul, Huqqani A., Schikuta E. (2009). Aggregating hierarchical service level agreements in business value networks. In *Proceedings of BPM 2009*, LNCS 5701, Germany, September, 2009.

Kikuchi S., Shimamura H., Kanna Y. (2007). Monitoring method of cross-sites' processes executed by multiple WS-BPEL processors. In *Proceedings of 9th IEEE CEC/EEE*, Japan, July, 2007.

Lohmann N., Wolf K. (2009). *Realizability is controllability.* In *Proceedings of 6th International Workshop on Web Services and Formal Methods*, Italy, September, 2009.

Luckham D. C. (2002). *The power of events: An introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Longman Publishing, Boston, MA, USA.

Moser O., Rosenberg F., Dustdar S. (2010). Event driven monitoring for service composition infrastructures. In *Proceedings of 11th WISE International Conference*, LNCS 6488, China, December, 2010. Springer.

Polyvyanyy A., Vanhatalo J., Völzer H. (2010). Simplified computation and generalization of the refined process structure tree. In *Proceedings of the 7th International conference on Web Services and Formal Methods.*, USA, September, 2010.

Subramanian S., Thiran P., Narendra N., Mostefaoui G., Maamar Z. (2008). On the enhancement of BPEL engines for self-healing composite web services. In *Proceedings of the International Symposium on Applications and the Internet, SAINT*, Finland, August, 2008.

Vanhatalo J., Volzer H., Leymann F. (2007). Faster and more focused control-flow analysis for business process models through SESE decomposition. In *Proceedings of Fifth International Conference on Service-Oriented Computing - ICSOC*, Austria, September, 2007.

Weidlich M., Polyvyanyy A., Mendling J., Weske M. (2010). Efficient computation of causal behavioural profiles using structural decomposition. In *Proceedings of 31st International Conference on Applications and Theory of Petri Nets, PETRI NETS*, LNCS 6128, Portugal, June, 2010.

Weidlich M., Ziekow H., Mendling J., Günther O., Weske M., Desai N. (2011). Event-based monitoring of process execution violations. In *Proceedings of BPM 2011*, LNCS 6896, France, August, 2011.

Wetzstein B., Karastoyanova D., Kopp O., Leymann F., Zwink D. (2010). Cross-organizational process monitoring based on service choreographies. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2010*, Switzerland, March, 2010.