Surveillance de compositions de web services - Deux approches distribuées à base de chroniques pour la surveillance et le diagnostic

Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, Laurence Rozé

Surveillance de compositions de web services

Deux approches distribuées à base de chroniques pour la surveillance et le diagnostic

Xavier Le Guillou* — **Marie-Odile Cordier*** — **Sophie Robin*** — **Laurence Rozé****

* *Université de Rennes 1* ** *INSA de Rennes*
IRISA - Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cédex – FRANCE
prénom.nom@irisa.fr

RÉSUMÉ. *La capacité à créer des logiciels auto-diagnosticables et auto-réparables est un véritable défi pour la recherche à venir. Cet article décrit une architecture destinée à la surveillance et au diagnostic de web services. L'une des principales difficultés dans ce contexte est que les pannes se propagent d'un service à l'autre, ce qui fait du diagnostic une étape important pour réagir de manière pertinente. Notre principale contribution est d'étendre l'approche de reconnaissance de chroniques, reconnue comme efficace pour surveiller des systèmes industriels, à un contexte distribué tel que celui des chorégraphies de web services. Nous étudions deux cas, celui où les interactions entre services sont statiques et décrites au départ en WS-CDL et celui où le modèle des interactions est dynamique et doit être construit en ligne. Ce travail a été réalisé et financé dans le cadre du projet européen* WS-DIAMOND.

ABSTRACT. *The ability to create self-healing software is a challenging task for research. This article describes an architecture dedicated to the monitoring and the diagnosis of web services. One of the main difficulties in this context is that faults may propagate from a service to another, which makes of diagnosis a crucial issue in order to react properly. Our main contribution is to extend the chronicle recognition approach to a distributed context such as choreographies of web services. Two cases are studied. In the first one, interactions between services are static and described a priori in WS-CDL; in the second one, the model of interactions is dynamic and built online. This work has been achieved and funded within the framework of the* WS-DIAMOND *European project.*

MOTS-CLÉS : *diagnostic, monitoring, chroniques, web services, chorégraphies*

KEYWORDS: *diagnosis, monitoring, chronicles, web services, choreographies*

RSTI - RIA. Volume 24 – n°2/2010, pages 189 à 225

1. Introduction

Un paradigme émergent de l'informatique orientée service (Papazoglou *et al.*, 2003) repose sur la combinaison de web services existants dans le but de fournir des services à valeur ajoutée. Un défi important conditionnant une utilisation réelle des web services consiste à surveiller leur exécution et à les rendre capables de réagir à des dysfonctionnements imprévus grâce à des stratégies de récupération sur erreur.

Le langage de processus d'entreprises WS-BPEL (Alves *et al.*, 2007) fournit des mécanismes de compensation permettant de réagir de manière prédéfinie à des problèmes locaux spécifiques et prévus. Cependant, dans des environnements dynamiques tels qu'Internet, les web services peuvent être sujets à des dysfonctionnements imprévus pour lesquels il n'est pas possible de définir des mécanismes de détection ou des stratégies de récupération lors de la phase de conception. De plus, la gestion locale des fautes ne tient pas compte des partenaires du service, ce qui limite l'efficacité des stratégies de récupération sur erreur. Les cas difficiles concernent les pannes se propageant à travers les services avant d'être détectées. Dans un tel cas, le système ne revient dans un état normal que lorsque la cause primaire du dysfonctionnement est localisée et le service identifié comme fautif réparé.

Il est ainsi important de surveiller le traitement de la requête et d'effectuer un diagnostic dans le but de localiser les services fautifs, d'identifier l'erreur et de réagir correctement, afin que la requête soit traitée avec la meilleure qualité de service possible.

Considérons par exemple un contexte de *e-commerce* et supposons qu'un client commande une caisse de champagne brut et une boîte de foie gras via une boutique en ligne. La boutique envoie une requête pour chacun des produits commandés aux web services des fournisseurs spécialisés choisis, en leur transmettant un code produit. Dans le cas où la base de données du fournisseur de champagne n'est pas cohérente avec celle de la boutique, le colis va être préparé avec un autre produit, du mousseux par exemple. Tant qu'il n'y a pas mise en correspondance entre les libellés des produits commandés et les produits du colis, le processus continue et il n'est pas possible de détecter le problème. Supposons maintenant que le colis soit contrôlé avant expédition et que l'on découvre l'erreur (c'est un symptôme), il est possible, et important pour la suite, d'effectuer un diagnostic dans le but de localiser les services fautifs (ici le fournisseur de champagne), d'identifier l'erreur (ici la base de données du fournisseur) et de réagir, en modifiant la base de données erronnée mais aussi en modifiant le contenu du colis, éventuellement avec un surcoût, afin que la requête courante et les suivantes soient traitées avec la meilleure qualité de service possible.

Dans cet article, nous proposons une approche de diagnostic à base de modèle visant à gérer les dysfonctionnements imprévus à l'exécution et les conséquences d'une propagation de pannes. Ce travail a été réalisé dans le cadre du projet européen WS-DIAMOND (Web Services DIAgnosability, MONitoring and Diagnosis) (The WS-Diamond Team, 2007). Nous nous concentrons ici sur l'exécution d'une seule instance d'un service. Des travaux proposant d'agréger les informations prove-

nant d'un ensemble d'instances peuvent être trouvés dans (Ben Halima *et al.*, 2008). Nous nous concentrons également sur les fautes sémantiques, qui mènent à des résultats inattendus mais peuvent difficilement être détectées par le service sur lequel elles se produisent.

Nous proposons d'ajouter un diagnostiqueur local à chaque service, afin d'analyser les traces d'exécution, de détecter les dysfonctionnements et d'effectuer un diagnostic local. Un diagnostiqueur global est alors en charge de coordonner les diagnostiqueurs locaux et de calculer un diagnostic identifiant le service (et l'activité) supposé responsable de la faute.

L'approche que nous adoptons repose sur la reconnaissance de chroniques, déjà utilisée pour la surveillance de systèmes industriels complexes. Une chronique décrit une situation à surveiller sous la forme d'un ensemble d'évènements temporellement contraints. Notre contribution consiste à étendre l'approche existante au contexte spécifique des web services, la difficulté principale étant de gérer des systèmes hautement distribués dont le modèle des interactions est en général dynamique et non nécessairement connu à l'avance. Deux plates-formes ont été conçues. La première, nommée MATRAC, est dédiée aux chorégraphies de web services dont le modèle des interactions est statique et décrit dans un fichier WS-CDL. La seconde, nommée CARDECRS, supporte les compositions dont le modèle des interactions n'est pas connu a priori mais est construit dynamiquement en ligne.

Cet article est organisé comme suit. Nous commençons, en section 2, par présenter l'exemple d'e-commerce qui sera utilisé tout au long de cet article afin de motiver et d'illustrer notre propos. La section 3 pose le problème de diagnostic qui fait l'objet de l'article. La section 4 décrit la modélisation sur laquelle s'appuie notre approche et introduit le formalisme des chroniques communicantes. En section 5, nous étendons l'approche par reconnaissance de chroniques aux contextes distribués et donnons une vision à haut niveau du protocole utilisé pour calculer le diagnostic global à partir des diagnostics locaux. Les deux approches développées sont présentées, d'abord celle où le modèle des interactions est construit en ligne, puis celle s'appuyant sur un modèle des interactions décrit en WS-CDL. En section 10, nous comparons notre approche à des travaux proches, puis nous concluons en section 11.

2. Application

2.1. *Introduction de l'exemple applicatif*

Dans toute la suite de l'article nous utiliserons un même exemple de e-commerce. Ce web service, noté par la suite WS, résulte de la composition de trois web services : une boutique (Shop ou SH), un fournisseur (Supplier ou SU) et un entrepôt (Warehouse ou WH). L'interaction entre ces services est résumée par la figure 1.

Lorsqu'une boutique reçoit la commande qu'un client a passée en ligne, elle la décompose en deux : la partie de la commande portant sur des produits disponibles

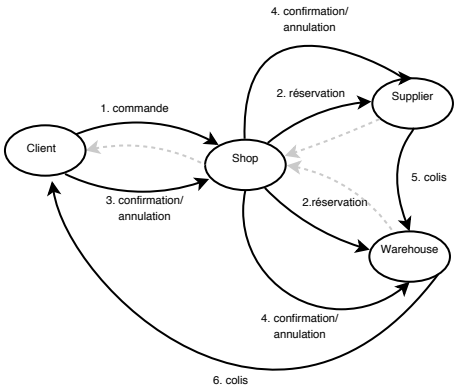


Figure 1. *Interactions entre les participants de l'exemple d'e-commerce*

auprès d'un fournisseur et la partie portant sur des produits stockés dans l'entrepôt. La boutique transmet ensuite les deux parties de commandes au fournisseur et à l'entrepôt dans le but de réserver les produits correspondants. En retour, elle reçoit de chacun d'eux la liste des produits disponibles. Elle transmet la liste des produits disponibles au client et attend la confirmation ou l'annulation du client. Si le client confirme, un message de confirmation est envoyé à la fois au fournisseur et à l'entrepôt. Le fournisseur prépare alors le colis et l'envoie à l'entrepôt. Ce dernier complète le colis en y ajoutant les produits qui le concernent et envoie le colis au client. La figure 2 résume ce fonctionnement normal de la composition de services qui nous sert d'exemple. On peut remarquer que le cas où le client ne confirme pas la commande, et où il faut annuler la réservation faite, n'est pas traité, ni la partie traitant le paiement de la commande par le client. Cet exemple est inspiré de celui traité dans le projet européen WS-DIAMOND.

2.2. *Pannes et symptômes*

Un certain nombre de pannes peuvent survenir sur ce web service, pannes dont le traitement n'est pas intégré dans le code (BPEL) des services. Ces pannes sont des événements exogènes qui sont de différents types : elles peuvent résulter de pannes matérielles, d'erreurs humaines, ou de problème de base de données[1].

Une panne se manifeste en général par un certain nombre de symptômes. On appelle *symptôme* un événement observable qui révèle un dysfonctionnement dû à une panne. Un symptôme est observé après l'occurence de la panne et éventuellement dans un service qui n'est pas celui où s'est produit la panne lorsqu'il y a propagation de la

1. Rappelons que nous ne considérons pas les pannes logicielles (erreurs de programmation) qui posent des problèmes de débuggage de programmes tout à fait différents.

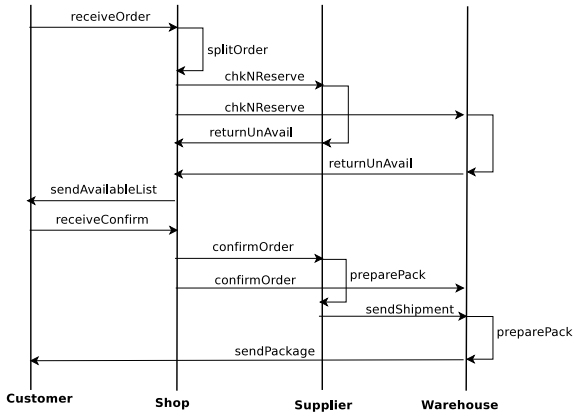


Figure 2. *Un scénario normal de l'exemple d'e-commerce*

panne à d'autres services avec lequel le premier communique. Les deux principaux types de symptômes sont la non-conformité entre une observation normalement attendue et l'observation faite ou un timeOut, c'est-à-dire un délai anormalement élevé (supérieur à un seuil) entre une requête et sa réponse.

Afin de pouvoir les diagnostiquer et y réagir avec pertinence, il est nécessaire que la liste de pannes soit spécifiée au départ ainsi que leurs symptômes, comme il est couramment admis dans les approches utilisant une approche à base de modèles de panne. Nous présentons ci-dessous les pannes que nous considérons pour l'application qui nous sert d'exemple et y associons les symptômes permettant de les détecter. On verra plus précisément dans la section 4 le mode de représentation de ces informations.

Les pannes présentées dans cette section sont représentées sur les diagrammes d'activité de la composition de services en figure 4. Les pannes relevant d'erreurs humaines incluent les erreurs de frappe. Le client peut faire une erreur de frappe dans son adresse mail (`SHmailFault`) ou son adresse postale (`SHaddressFault`). Si le client se trompe d'adresse postale, il ne recevra jamais son colis, et finira par envoyer une réclamation à la boutique. Le symptôme, détecté au niveau de la boutique, est appelé `SHtimeOutReceptPackage`. Si le client se trompe d'adresse mail, il ne recevra pas la liste des produits disponibles et n'enverra donc pas de confirmation. Trois symptômes de type timeOut (`timeOutConfirm`) seront observés au niveau de la boutique, du fournisseur et de l'entrepôt.

Des pannes matérielles peuvent se produire du côté du fournisseur ou de l'entrepôt : un incident technique peut faire que le colis n'est jamais envoyé. Si cet incident se produit dans l'entrepôt (`WHhardFault`), le client ne va jamais recevoir son colis et va donc aussi envoyer une réclamation à la boutique (`SHtimeOutReceptPackage`). Si la panne matérielle se produit chez le fournisseur (`SUhardFault`), deux symptômes vont apparaître, l'un du côté de l'entrepôt (`WHtimeOutReceptPackage`)

qui attend de recevoir le colis du fournisseur et l'autre du côté de la boutique (`SHtimeOutReceptPackage`) lors de la réception de la réclamation client.

Des pannes dues à des erreurs dans les bases de données peuvent se produire chez le fournisseur ou à l'entrepôt, par exemple si leur base de données Produit n'est pas cohérente avec celle de la boutique. C'est alors un mauvais produit qui est ajouté au colis. Si cette panne se produit dans l'entrepôt (`WHreserveFault`), le client va recevoir un colis dont un des produits ne correspond pas à sa demande, et le symptôme est la réclamation qu'il envoie à la boutique (`SHbadPackage`). Si l'erreur de réservation se produit du côté du fournisseur (`SUreserveFault`), un premier symptôme va apparaître puisque le contenu du colis est vérifié par l'entrepôt qui ajoute une note (`WHbadPackage`) au colis. Le second symptôme correspond à la réclamation que le client envoie à la boutique (`SHbadPackage`).

Enfin des pannes dues à des problèmes de stock informatique peuvent se produire : le produit est enregistré comme disponible, mais n'est pas en réalité en stock (`WHstockFault` et `SUstockFault`). Ce problème donne lieu à un symptôme (`badPackage`) au niveau de l'entrepôt ou du fournisseur selon le cas, avec ajout d'une note dans le colis. puis à l'envoi du colis au client. Ensuite, le client envoie, à la réception de son colis, un mail de réclamation à la boutique (`SHbadPackage`).

3. Problème de diagnostic

Après un bref rappel de ce qu'est un problème de diagnostic à base de modèles, nous le définissons plus précisément dans le cadre distribué qui est celui d'un ensemble de services à surveiller. Pour traiter ce problème, nous proposons d'utiliser une approche de type reconnaissance de chroniques et en rappelons les éléments essentiels mais sans rentrer dans le détail de cette approche supposée connue (voir (Dousson *et al.*, 1993)).

Le problème de diagnostic consiste, à partir d'un ensemble d'observations, à détecter un fonctionnement anormal de ce système, puis à localiser et caractériser la source de ce dysfonctionnement. Le diagnostic est en général fait en vue d'une action de remédiation, qu'elle soit décidée par un opérateur ou déclenchée automatiquement. Cette action peut être la réparation ou le remplacement d'un composant du système, une reconfiguration, ou toute autre action permettant le retour à un bon fonctionnement. Dans cet article, nous nous intéressons à la surveillance d'un système dynamique en ligne ; les observations sont donc acquises au fil du temps et le diagnostic analyse le flux d'informations acquises sur le système. De plus, nous nous focalisons sur la partie diagnostic sans aborder la partie réparation.

3.1. *Système et modèle distribué*

Pour effectuer la tâche de surveillance et de diagnostic de web services, nous avons choisi une approche de diagnostic à base de modèles, dont la caractéristique principale

est de s'appuyer sur une modélisation du système à surveiller. Dans le cadre des web services, le système à surveiller est par nature un système distribué puisqu'il s'agit d'un ensemble de services communiquant entre eux par l'intermédiaire de requêtes afin de satisfaire de manière coordonnée la requête d'un utilisateur. Le modèle du système est un modèle distribué qui comporte d'une part le modèle de chacun des composants (ou services) et d'autre part le modèle des interactions qui décrit la manière dont ces services communiquent entre eux et se synchronisent. Cette modélisation est décrite plus précisément dans la section suivante et est notée $\mathcal{M}$.

3.2. *Pannes et propagation des pannes*

Dans un système dynamique et distribué, la tâche de diagnostic consiste à surveiller le comportement de chacun des composants constituant le système afin de détecter et d'identifier un dysfonctionnement. On appelle *panne* ($fault$ en anglais), un dysfonctionnement qui se produit dans un des composants et peut par propagation provoquer des dysfonctionnements, éventuellement observables, dans des composants avec lesquels ce composant, qui se trouve à la source du problème, communique. Un composant peut très bien avoir un comportement normal mais transmettre un dysfonctionnement à un autre composant, en particulier lorsque la panne affecte des informations qui transitent au travers de services comme dans le cas de web services. C'est ce que l'on appelle en général les pannes en cascade. Nous réservons dans la suite le terme de panne au problème qui est à la source des dysfonctionnements qu'elle peut provoquer par propagation. L'ensemble des pannes est noté $\mathcal{P}$.

Dans le cadre des web services, nous nous sommes intéressés aux pannes portant sur les données, pannes dites sémantiques, comme une mauvaise correspondance entre un code et un libellé, une mauvaise cohérence entre plusieurs bases de données, une entrée erronée de la part de l'utilisateur, ainsi que les pannes dites humaines, comme une erreur lors de la constitution d'un colis. Ces pannes ne sont donc pas des pannes système telles que pourrait l'être une panne d'un serveur dans notre cas. En général, ce type de pannes sont rarement détectables dans le service où elles se produisent, et se propagent avant détection. Le diagnostic nécessite donc d'avoir une vue globale de la circulation des informations et on ne peut se contenter d'une étape locale de détection/diagnostic de type traitement des exceptions. Nous nous limitons actuellement au cas de la panne unique.

3.3. *Observations*

La tâche de diagnostic s'appuie sur un ensemble d'informations acquises sur le système, appelées *observations* et notées $\mathcal{O}$. Dans le cas de web services, les observations sont naturellement distribuées dans chacun des services. Nous supposons que chaque service stocke, sous forme de logs, une séquence d'événements correspondant aux activités exécutées au fil du temps. Nous supposons pour l'instant que l'ensemble des événements correspondant aux activités d'un service sont observables. Parmi ces

observables, certains, que nous avons appelés symptômes dans la section 2, sont provoqués par des pannes et sont donc utiles pour identifier les pannes que l'on souhaite surveiller[2].

3.4. *Approche de diagnostic*

Le problème de diagnostic que nous abordons est défini par le triplet $(\mathcal{M}, \mathcal{O}, \mathcal{P})$ et consiste à identifier une panne, élement de $\mathcal{P}$, à partir du modèle distribué du système $\mathcal{M}$ et des observations $\mathcal{O}$ stockées localement par les services décrits par $\mathcal{M}$. L'approche retenue est une approche distribuée dont l'architecture est illustrée par la figure 3. Elle s'appuie sur des diagnostiqueurs locaux, en charge de détecter des dysfonctionnements au niveau des services, à partir des logs d'observations, et de calculer des diagnostics locaux. En raison de la propagation des pannes, il est en général nécessaire de faire appel à un diagnostiqueur global pour raffiner les diagnostics locaux. Ce diagnostiqueur global utilise la connaissance qu'il a des interactions entre services (modèle des interactions) pour, par un mécanisme de type push-pull, décrit plus en détail en section 5, confronter les diagnostics locaux, éliminer les hypothèses impossibles et calculer le diagnostic global.

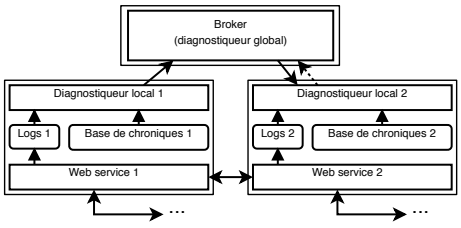


Figure 3. *Architecture générale de notre approche de diagnostic*

L'approche choisie de diagnostic (en particulier au niveau des services) est une approche à base de modèles dite reconnaissance de chroniques. Cette approche est efficace et a été déjà largement utilisée pour la surveillance de systèmes industriels complexes. Nous disposons de l'outil CRS développé par France Telecom R&D. Cette approche consiste à associer à chaque situation à surveiller (et donc chaque panne) un motif permettant de l'identifier. Le flux d'observations est alors analysé efficacement et les motifs reconnus en ligne. Une chronique décrit une situation à surveiller sous la forme d'un ensemble d'évènements observables temporellement contraints. Le diagnostic local sera ainsi calculé par un diagnostiqueur local en s'appuyant sur un reconnaisseur de chroniques CRS local et utilisant une base de chroniques locales.

2. Il serait possible de ne stocker dans les logs que les observables qui participent à l'identification des pannes que l'on souhaite surveiller. Mais ceci nécessite une analyse approfondie de diagnosticabilité du système (Sampath *et al.*, 1995), ou même de *self-healability* (Cordier *et al.*, 2008), dont nous ne tenons pas compte dans cet article.

Notre contribution principale consiste à étendre l'approche de reconnaissance de chroniques au contexte distribué des web services. Après une section consacrée à la modélisation, distinguant la modélisation du comportement de chacun des services (comportement normal et anormaux) et la modélisation de l'interaction entre les différents services, nous décrirons plus en détail en section 5, la manière dont nous avons résolu ce problème de diagnostic distribué.

4. Modélisation

La modélisation de la composition de services web que nous considérons se compose d'un modèle de comportement, rassemblant les modèles de comportement des services web impliqués ainsi qu'un modèle des interactions qui décrit les échanges entre les services impliqués dans la composition.

Nous considérons des environnements dans lesquels un workflow de chaque service web impliqué dans la composition est disponible. Ce workflow peut être construit par exemple à partir du code BPEL *(Web Services Business Process Execution Language)* qui est devenu un des standards dans la description de services web. Le modèle de comportement se dérive naturellement de cet ensemble de workflows, au moins en ce qui concerne le comportement normal des services. Nous allons voir comment ce modèle est enrichi pour tenir compte des comportements anormaux en 4.1.2. En ce qui concerne la description des interactions entre les services, deux cas se présentent. Dans le cas où les échanges entre services sont décidés en ligne, et donc où la composition est dynamique, ce qui devrait être le cas dans l'univers des services web, le modèle des interactions ne peut qu'être construit lui aussi dynamiquement. En revanche, dans le cas où les interactions entre services sont connues au départ, le modèle des interactions est statique, et on peut imaginer en disposer explicitement sous la forme d'une description WS-CDL *(Web Services Choreography Description Language)* qui représente, du point de vue d'un observateur global, la chorégraphie des messages ordonnés échangés.

4.1. *Modèle de comportement*

Le modèle de comportement est composé des modèles de comportement normal (ou attendu) de chacun des services de la composition et il est enrichi du modèle de leurs comportements anormaux, en fonction des pannes que l'on désire surveiller.

4.1.1. *Modèle de comportement normal*

Les services web que nous considérons sont modélisés par des workflow dans lesquels les briques de base sont des activités qui peuvent être :

– des *activités internes* telles que « le service compare les valeurs des variables x_1 et x_2 »

– des *activités d'envoi ou de réception de messages* telles que « le service invoque un service partenaire avec les paramètres x, y et z »

Ces briques de base sont alors combinées par l'utilisation de structures de contrôle telles que séquence, choix multiple ou encore boucle. Plus précisément, nous avons retenu les workflow patterns 1 à 7 et 21 parmi ceux introduits par Van Der Aalst (van der Aalst *et al.*, 2003) parce qu'ils sont bien représentatifs des processus que nous voulons modéliser. Sur la figure 4 sont représentés les diagrammes d'activité des trois services (boutique, fournisseur, entrepôt) impliqués dans l'exemple introduit à la section 2.

4.1.2. *Modèle enrichi par les comportements anormaux*

Nous nous intéressons aux dysfonctionnements qui ne sont pas gérés par les mécanismes d'exception intégrés dans le code des services. Comme expliqué dans la section 2.2, nous considérons principalement les pannes dues à des erreurs sur les données d'entrée, à des erreurs humaines de manipulation ou des pannes matérielles. Ces pannes se propagent et sont à la source d'éventuels dysfonctionnements dans des services non directement en panne. Comme l'illustre la figure 4, chacun des diagrammes d'activité est enrichi pour chacune des pannes considérées. Les pannes sont représentées par un pentagone associé à l'activité « source » du dysfonctionnement. Les symptômes des pannes sont représentés par une étoile explosante associée à l'activité détectant le symptôme.

Cet enrichissement est fait manuellement par un expert. Il semble difficile de faire autrement en ce qui concerne les pannes puisqu'il faut indiquer les pannes qui feront l'objet de surveillance. En revanche, des travaux sont en cours pour acquérir automatiquement les symptômes nécessaires et suffisants pour identifier la panne ; ces travaux s'appuient sur les études de diagnosticabilité (Sampath *et al.*, 1995; Cordier *et al.*, 2008).

Les pannes retenues ici sont les suivantes : deux pannes `SHmailFault` et `SHaddressFault` au niveau du service client (SHop), correspondant à des erreurs de l'interface avec le client ; trois pannes `SUreserveFault`, `SUhardFault` et `SUstockFault` au niveau du service fournisseur (SUpplier) et trois pannes `WHreserveFault`, `WHhardFault` et `WHstockFault` au niveau du service entrepôt (WareHouse). Les symptômes sont de type `badPackage`, au niveau de la boutique (`SHbadPackage`), au niveau du fournisseur (`SUbadPackage`) et au niveau de l'entrepôt (`WHbadPackage`), correspondant à une erreur dans le colis (produit manquant ou erroné) ainsi que des symptômes de type time-out : `timeOutConfirm` au niveau de la boutique, du fournisseur, et de l'entrepôt en cas d'absence de confirmation de la part du client ; et `timeOutReceptPackage` au niveau de la boutique en cas de colis non reçu par le client et au niveau de l'entrepôt en cas de colis non envoyé par le fournisseur. Une explication de ces symptômes et de leurs liens avec les pannes est donnée dans la section 2.2.

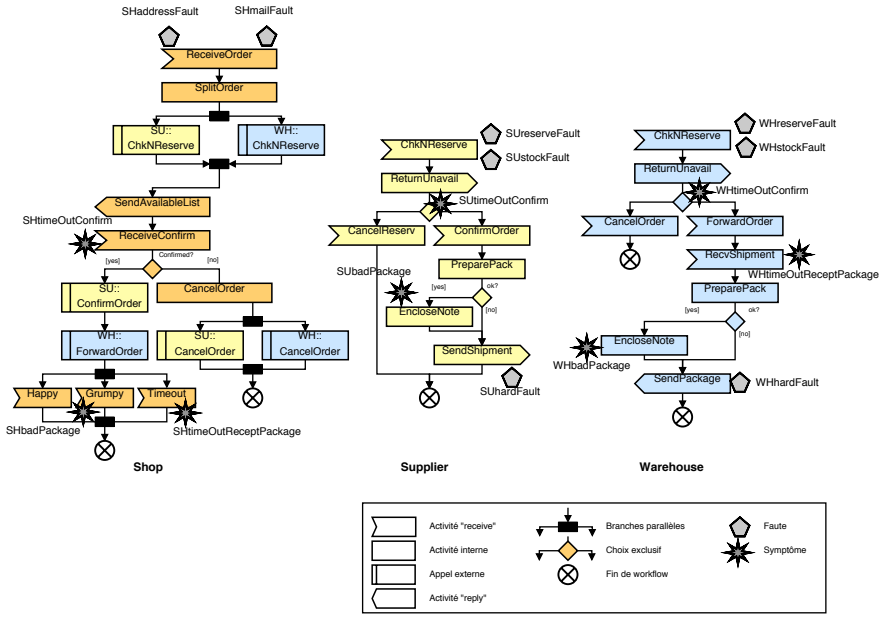


Figure 4. *Diagrammes d'activité enrichis de la boutique, du fournisseur et de l'entrepôt*

4.1.3. *Chroniques de comportement*

Nous avons choisi d'utiliser l'approche reconnaissance de chroniques (Dousson *et al.*, 1993) pour réaliser la tâche de surveillance et diagnostic de nos services web. Dans ce cadre, les comportements des services doivent être représentés dans le formalisme des chroniques. Rappelons qu'une chronique est un ensemble d'évènements observables, temporellement contraints et caractéristiques d'une situation.

Un comportement normal d'un service correspond à un chemin d'exécution dans le workflow du service et donc à un ensemble d'activités contraintes temporellement. Une fois le workflow enrichi par les pannes, comme indiqué à la section précédente, un comportement anormal se traduit lui aussi par un chemin d'exécution dans ce workflow enrichi. Nous considérons les activités comme des évènements et les supposons toutes observables (voir section 3.3).

Un **type d'évènement** définit ce qui est observé dans le système, par exemple dans notre cas le nom d'une activité act, le nom agrémenté du fait que l'activité débute (act^-) ou finit (act^+), le nom enrichi de paramètres observables $act(?var_1, \ldots, ?var_n)$ ou une combinaison de ces possibilités. $\mathcal{E}$ représente l'ensemble des types d'évènements possibles.

Un **évènement** est une paire $(e, ?t)$ où $e \in \mathcal{E}$ est un type d'évènement et $?t$ la date d'occurrence de l'évènement.

Une **chronique** $\mathcal{C}$ est une paire $(\mathcal{S}, \mathcal{T})$ où $\mathcal{S}$ est un ensemble d'évènements et $\mathcal{T}$ un ensemble de contraintes sur les dates d'occurrence. Une **instance de chronique** est une chronique dont les variables et leurs dates d'occurrence sont instanciées.

La chronique décrivant le comportement du fournisseur dans le cas de fonctionnement normal de la composition de services qui nous sert d'exemple (cf figure 4) est $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ où :

$$\begin{aligned}\mathcal{S} = \{ \quad & (ChkNReserve(?SUlistIn), ?t_1),\\ & (ReturnUnavail(?SUlistOut), ?t_2),\\ & (ConfirmOrder, ?t_3),\\ & (PreparePack, ?t_4),\\ & (SendShipment(?SUshipmentOut, ?t_5),\\ \}\end{aligned}$$

$$\mathcal{T} = \{?t_1 < ?t_2, ?t_2 < ?t_3, ?t_3 < ?t_4, ?t_4 < ?t_5\}$$

4.2. *Modèle des interactions*

La figure 5 représente l'ensemble des interactions entre les services dans l'exemple décrit à la section 2. Une interaction correspond à un échange entre deux services (un producteur et un consommateur) et va correspondre à deux points d'interaction (un sur chacun des deux services).

L'interaction 2 entre la boutique (Shop) et le fournisseur (Supplier) correspond à l'envoi de la liste des produits au fournisseur pour qu'il les réserve (au cas où ils sont disponibles). Cette interaction est effectuée par la requête de l'activité `SU::ChkNReserve` du service Shop à travers la variable `?SHlistSUOut`. Cette requête est reçue par l'activité `ChkNReserve` du service Supplier à travers la variable `?SUlistIn`.

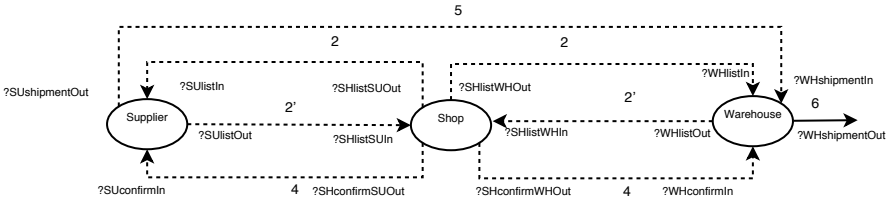


Figure 5. *Interactions entre la boutique, le fournisseur et l'entrepôt*

Une interaction correspond à deux points d'interaction. Un **point d'interaction** est défini comme un tuple $(inter_{id}, \{vars\}, serv_{id})$ où $inter_{id}$ est l'identifiant du point

d'interaction (par exemple un nom d'activité), $\{vars\}$ une liste ordonnée de variables échangées et $serv_{id}$ un identifiant correspondant au service distant partenaire.

Un point d'interaction est dit **entrant** s'il correspond à une communication orientée dans le sens $serv_{remote} \rightarrow serv_{local}$, **sortant** dans le cas contraire.

Ainsi, le point d'interaction $(SU :: ChkNReserve, (?SHlistSUOut), Supplier)$ est un point d'interaction du service Shop vers le service Supplier, $(ChkNReserve, (?SUlistIn), Shop)$ un point d'interaction entrant du service Shop vers le service Supplier.

Une **instance de point d'interaction** est un point d'interaction dont les variables ont été instanciées et dont l'identifiant $serv_{id}$ a pris pour valeur l'adresse effective du service partenaire.

Une **interaction** est alors modélisée par un couple (pi_s, pi_e) où pi_s est un point d'interaction sortant et pi_e un point d'interaction entrant. pi_s et pi_e sont qualifiés de points d'interaction **homologues**. Une interaction permet d'établir la correspondance entre des variables de services différents mais impliquées dans le même échange.

L'interaction 2 entre la boutique et le fournisseur est modélisée par le couple :
$((SU :: ChkNReserve, (?SHlistSUOut), Supplier),$
$(ChkNReserve, (?SUlistIn), Shop))$
le premier point d'interaction correspondant à un point d'interaction sortant de la boutique et le second à un point d'interaction entrant sur le fournisseur.

Le **modèle des interactions** d'une composition de services web est alors décrit comme un ensemble d'interactions.

4.3. *Chroniques communicantes*

La reconnaissance de chroniques est une approche de diagnostic classique dans le cas centralisé. Nous avons choisi de l'utiliser dans le cas distribué et une de nos contributions est d'étendre le formalisme des chroniques classiques afin de prendre en compte les interactions entre composants et permettre d'exprimer des contraintes (dites souvent contraintes de synchronisation) que le diagnostiqueur global devra vérifier. Dans cette section, nous présentons le formalisme des chroniques communicantes qui étend le formalisme des chroniques.

Une chronique communicante décrit un des comportements possibles d'un service web en ajoutant à la chronique de comportement vue précédemment la description des interactions de ce service avec les services partenaires ; ceci est en particulier important pour décrire les effets d'une panne par propagation. Deux comportements d'un service peuvent différer soit par l'ensemble des activités (le chemin d'exécution emprunté dans le workflow), soit par le statut des variables échangées lors des activités de communication (d'échange) avec un autre service web. C'est notamment le statut des variables échangées qui va différencier un comportement de panne d'un compor-

tement normal. Par exemple, pour le service Shop, dans le cas d'une erreur de réservation par le service Supplier (`SUreserveFault`), la suite d'activités effectuée est la même que dans le cas d'un comportement normal du service ; les deux chroniques seront donc reconnues ensemble par le diagnostiqueur local associé au service Supplier. Par contre, c'est le statut de la variable `?SUshipmentOut` (cf. figure 5) qui diffère. Dans le cas d'une infection du service Shop par le service Supplier, la valeur de cette variable n'est pas conforme à la valeur attendue dans le cas d'un comportement normal et son statut est alors dit erroné. C'est le statut de cette variable qui permettra au diagnostiqueur global de faire la différence entre les deux chroniques.

Le **statut d'une variable** est un booléen indiquant si, dans un contexte d'exécution donné, la valeur de la variable de chronique est conforme à la valeur attendue dans le cas d'un comportement normal ($\neg err$) ou non (err).

Un **point de synchronisation** est alors un couple (pi, ls) où pi est un point d'interaction et ls la liste des statuts associés à chacune des variables échangées. En référence à la figure 4 et à la section 2, voici l'instanciation d'un des points de synchronisation du service Supplier (fournisseur), dans le cas d'exécution d'une erreur de réservation : $((SendShipment, (?SUshipmentOut), Warehouse), (err))$. Elle exprime le fait que la variable `?SUshipmentOut`, que le service Supplier transmet au service Warehouse lors de l'activité `SendShipment`, est erronée.

La tâche de diagnostic est distribuée et gérée dans le cadre d'un mécanisme de type push-pull (voir section 3). Chaque diagnostiqueur local s'appuie sur un ensemble de chroniques et doit pouvoir "réveiller" (push) le diagnostiqueur global, en cas de problème détecté localement. C'est le rôle de la **couleur d'une chronique** qui représente sa capacité à déclencher un processus de diagnostic global. Deux couleurs sont utilisées : $rouge$ pour les comportements considérés commes critiques et devant déclencher le diagnostiqueur global et $vert$ pour les comportements perçus comme normaux et les comportements de panne considérés comme non critiques. Par exemple, dans le cas d'une erreur de stock informatique où une note est ajoutée au colis, on peut considérer qu'il s'agit d'une panne non critique et attribuer la couleur verte à la chronique associée.

Une chronique communicante est une chronique classique enrichie d'une "couleur" et des éléments de "synchronisation", qui vont permettre de la mettre en relation avec les chroniques de services adjacents.

Une **chronique communicante** est un tuple $\mathcal{C}_c = (\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I}, \mathcal{K})$ où $\mathcal{N}$ est le nom (unique) de la chronique communicante, $\mathcal{S}$ est un ensemble d'évènements, $\mathcal{T}$ un graphe de contraintes entre les dates d'occurrence, $\mathcal{O}$ et $\mathcal{I}$ sont respectivement deux ensembles de points de synchronisation sortants et entrants, et $\mathcal{K}$ est la couleur de la chronique.

Considérons la chronique communicante suivante décrivant le cas d'une erreur de réservation sur le fournisseur. $\mathcal{C}_c = (\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I}, \mathcal{K})$:

$\mathcal{N} = SU :: reserveFault,$
$\mathcal{S} =$ $(ChkNReserve(?SUlistIn), ?t_1),$
$(ReturnUnavail(?SUlistOut), ?t_2),$
$(ConfirmOrder, ?t_3),$
$(PreparePack, ?t_4),$
$(SendShipment(?SUshipmentOut), t_5),$
$\mathcal{T} = \{?t_1 <?t_2, ?t_2 <?t_3, ?t_3 <?t_4, ?t_4 <?t_5\}$
$\mathcal{O} = \{((ReturnUnavail, (?SUlistOut), Shop), (\neg err)),$
$((SendShipment, (?SUshipmentOut), Warehouse), (err))\}$
$\mathcal{I} = \{((ChkNReserve, (?SUlistIn), Shop), (\neg err))\}$
$\mathcal{K} = vert$

L'ajout des informations de synchronisation à une chronique pour la rendre chronique communicante a pour but de pouvoir retrouver les instances de chroniques appartenant à un même chemin dans le workflow. Ceci permet ensuite de pouvoir, si besoin, fusionner deux chroniques compatibles. Deux chroniques sont homologues si elles correspondent à deux services différents mais partagent une même interaction. Deux chroniques homologues sont compatibles si elles correspondent à un même comportement, ce qui se traduit par le fait que, pour chacun de leurs points de synchronisation homologues, les variables échangées au travers du point d'interaction ont le même statut.

Deux points de synchronisation (pi_1, ls_1) et (pi_2, ls_2) sont **homologues** si les points d'interaction pi_1 et pi_2 sont homologues (section refmodele-inter). Deux chroniques communicantes sont dites **homologues** si tous leurs points de synchronisation correspondent à des points de synchronisation homologues deux à deux. Deux points de synchronisation homologues (pi_1, ls_1) et (pi_2, ls_2) sont dits **compatibles** si les deux listes ordonnées de statuts de variables ls_1 et ls_2 sont identiques. Deux chroniques communicantes homologues sont **compatibles** si tous leurs points de synchronisation sont compatibles.

5. Diagnostic local et global

La figure 6 résume l'architecture de notre approche basée sur les chroniques. Ce système décentralisé se compose d'un diagnostiqueur global (ou *broker*). Son rôle est de fusionner les diagnostics locaux envoyés par chaque service en vérifiant leur compatibilité et d'envoyer les diagnostics globaux à un module de réparation. Chaque service contient un web service à proprement parler, des logs stockant en temps réel les évènements observables, un diagnostiqueur local chargé d'analyser les logs et d'instancier les chroniques de la base de *chroniques communicantes* (générée hors ligne).

La section 5.1 décrit l'architecture locale et le fonctionnement d'un diagnostiqueur local. La section 5.2 décrit l'architecture globale et le fonctionnement général de l'algorithme de diagnostic.

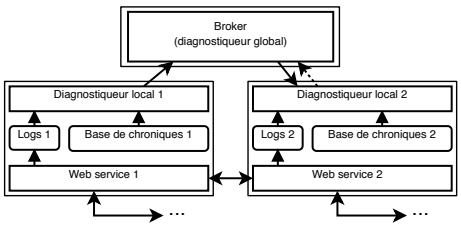


Figure 6. *Architecture générale de notre approche de diagnostic*

5.1. *Architecture locale et calcul du diagnostic local*

Le calcul du diagnostic local repose sur une reconnaissance de chroniques, représentant les différents comportements possibles du service, grâce à l'observation d'évènements s'étant produits. Les chroniques, une fois reconnues, sont mémorisées sous la forme d'abrégés de chroniques.

5.1.1. *Abrégés de chroniques communicantes*

Un abrégé de chronique communicante résume les informations d'une chronique communicante et ne contient que les informations nécessaires à la fusion avec les abrégés de chroniques communicantes reconnues sur des services partenaires.

Un **abrégé de chronique communicante** est défini comme un n-uplet $id_{evt}[id_c, \mathcal{K}, \{p_s\}]$ où id_{evt} est un identifiant unique, id_c est le nom de la chronique, $\mathcal{K}$ est la couleur de la chronique et $\{p_s\}$ l'ensemble des points de synchronisation instanciés de la chronique.

5.1.2. *Diagnostic local*

Comme l'illustre la figure 7, un diagnostiqueur local se compose d'un moteur de reconnaissance de chroniques et d'une base des chroniques à reconnaître. Le moteur, dans notre cas CRS, est chargé de scruter les logs d'exécution, d'instancier les modèles de chroniques de la base et de stocker dans un tampon les abrégés de chroniques reconnues (complètement instanciées).

La gestion du tampon dépend de la couleur des abrégés de chroniques qui y sont stockés. Le mécanisme de filtrage est le suivant :

– une chronique rouge est envoyée directement par le tampon au diagnostiqueur global ;

– une chronique verte n'est envoyée au diagnostiqueur global que si le tampon a reçu une commande (dite de focalisation) lui demandant de laisser passer les chroniques vertes ; sinon elle est stockée dans le tampon.

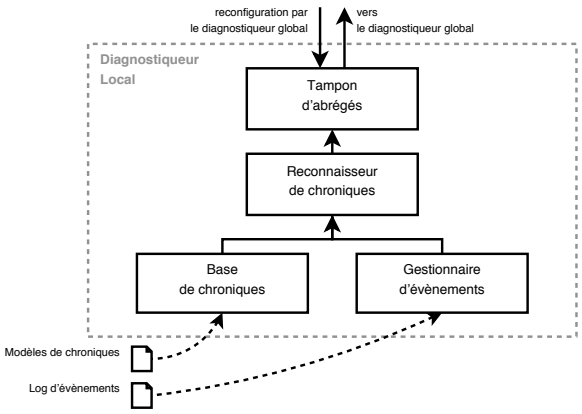


Figure 7. *Architecture détaillée d'un diagnostiqueur local*

L'utilisation du *tampon* du diagnostiqueur local par le diagnostiqueur global est décrite en section 5.2.2.

5.2. *Architecture globale et calcul du diagnostic global*

Le diagnostiqueur global s'appuie sur les diagnostics locaux calculés par les diagnostiqueurs locaux afin de calculer un diagnostic global expliquant le comportement observé sur le système. Afin d'être en mesure de fusionner ces différents diagnostics locaux, le diagnostiqueur global doit pouvoir retrouver les chroniques compatibles, ce qui implique une connaissance du modèle des interactions entre services (voir 4.2).

5.2.1. *Problème de la fusion des chroniques*

À l'exécution, plusieurs problèmes se posent afin d'être en mesure de fusionner des chroniques d'une manière cohérente :

– les chroniques ne sont susceptibles de fusionner que si elles appartiennent à des services ayant interagi durant l'exécution ;

– les chroniques fusionnées doivent correspondre à une seule et même instance d'exécution des services web (filtrage par identifiant de processus) ;

– chaque message envoyé par un service source doit fusionner avec le message correspondant reçu par le service cible.

On rappelle que le modèle des interactions permet de déterminer si deux chroniques communicantes (ou deux abrégés de chroniques communicantes) sont homologues et compatibles. Le modèle des interactions peut être soit construit dynamiquement en ligne (on verra en section 6 que cela nécessite le respect d'une méthodologie

d'écriture des chroniques), soit connu dès le départ par le diagnostiqueur global (section 7).

5.2.2. *Algorithme push-pull*

L'algorithme de diagnostic push-pull que nous utilisons est implémenté grâce à un mécanisme de filtrage/focalisation qui gère l'envoi des chroniques du diagnostiqueur local vers le diagnostiqueur global.

Il y a deux phases dans le processus de diagnostic global (figure 8). Dans une première phase (*push*), des diagnostiqueurs locaux envoient les abrégés de chroniques reconnues au *broker*, ce qui déclenche un calcul de diagnostic global. Dans une seconde phase (*pull*), c'est-à-dire lorsque le diagnostiqueur global a besoin d'informations complémentaires, il envoie une commande de focalisation aux diagnostiqueurs locaux leur demandant de lui envoyer les abrégés de toutes les chroniques qui ont été ou vont être reconnues.

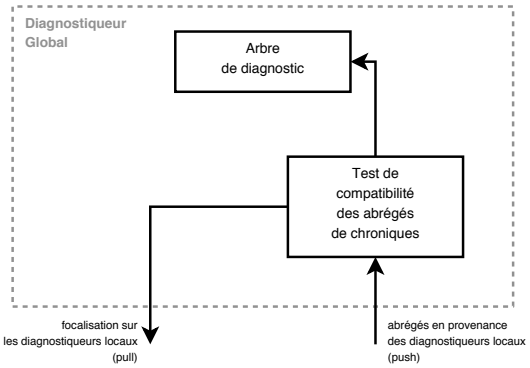


Figure 8. *Architecture détaillée du diagnostiqueur global*

5.2.2.1. Mécanisme de filtrage du diagnostiqueur local

Un diagnostiqueur local déclenche le diagnostiqueur global lorsqu'une chronique rouge est reconnue. Le *Tampon* (figure 7) est configuré au départ pour ne laisser passer que les abrégés de chroniques rouges et stocker les abrégés de chroniques vertes. Ultérieurement, la commande `focus(vert)` envoyée par le diagnostiqueur global peut purger le *Tampon* en libérant les abrégés des chroniques vertes qui sont alors insérés dans l'arbre de diagnostic. Ce mécanisme de filtrage est décrit par l'algorithme 1. $\mathcal{F}$ contient les couleurs des chroniques non filtrées. Cet ensemble est modifié dynamiquement. $\mathcal{C}_{buf}$ est le tampon (buffer) qui contient les abrégés de chroniques reconnues.

5.2.2.2. Mécanisme de focalisation du diagnostiqueur global

Lorsque le diagnostiqueur global reçoit un abrégé de chronique rouge de la part d'un diagnostiqueur local (*push*), une série d'opérations *pull* commence, par l'inter-

Init : $\mathcal{C}_{buf} := \emptyset$, $\mathcal{F} := \{rouge\}$
on event *abrégé* d_c *reçu* **do**
  if $(d_c.couleur \in \mathcal{F})$ **then** $relâcher(d_c)$;
  else $\mathcal{C}_{buf} := \mathcal{C}_{buf} \cup \{d_c\}$;
end
on event *focus(couleur* $\mathcal{K}$*)* **do**
  foreach *abrégé* $(d_c \in \mathcal{C}_{buf})$ **do**
    if $(d_c.couleur = \mathcal{K})$ **then**
      $relâcher(d_c)$;
      $\mathcal{C}_{buf} := \mathcal{C}_{buf} \backslash \{d_c\}$;
    end
  end
  $\mathcal{F} := \mathcal{F} \cup \{\mathcal{K}\}$;
end

Algorithme 1. *Diagnostiqueur local : filtrage et focalisation*

médiaire de la commande `focus(vert)` qu'il envoie à chaque diagnostiqueur local avec lequel il veut communiquer (figure 7).

Les abrégés de chroniques reçus par le diagnostiqueur global sont insérés dans un arbre de diagnostic $\mathcal{D}_t$, après vérification de leur compatibilité avec les abrégés de chroniques déjà stockées. Un nœud de l'arbre de diagnostic contient un ensemble d'abrégés de chroniques compatibles et un ensemble de contraintes de synchronisation restant à vérifier pour obtenir un diagnostic global. À chaque ajout d'un nœud dans l'arbre, les contraintes vérifiées par l'abrégé de chronique intégré sont supprimées. Un diagnostic global correspond ainsi à un nœud sans contrainte.

L'algorithme 2 décrit à haut niveau le fonctionnement du diagnostiqueur global à travers la greffe de nouveaux nœuds dans l'arbre de diagnostic et la focalisation sur les chroniques vertes.

Init : $\mathcal{D}_t := \{emptynode\}$
on event *abrégé* d_c *reçu* **do**
  foreach *instance* i *de* $\mathcal{D}_t$ **do**
    if d_c *compatible avec* i **then** $\mathcal{D}_t := \mathcal{D}_t \cup \{(i, d_c)\}$;
  end
  foreach *WebService* s *distant* **do** $s.focus(vert)$;
end

Algorithme 2. *Diagnostiqueur global : construction de l'arbre et focalisation*

Après avoir décrit de manière générique le fonctionnement de notre approche de surveillance et de diagnostic, nous examinons dans les deux sections suivantes les deux cas que nous avons développés et implémentés. Le premier (section 6) est ce-

lui où le modèle des interactions n'est pas connu à l'avance car il est dynamique ; il doit donc être construit en ligne. Le second (section 7) est le cas où le modèle des interactions est statique et est décrit dans un langage de type WS-CDL.

6. Diagnostiqueur global avec construction en ligne du modèle des interactions

Dans cette section, nous nous plaçons dans le cadre où les services web sont composés dynamiquement, via l'interrogation d'annuaires tels que les annuaires UDDI. La topologie des compositions de services surveillées n'est pas connue au préalable, il n'est donc pas envisageable d'établir a priori un modèle des interactions qui pourrait servir au diagnostic global. Il faut donc que le diagnostiqueur global le construise en ligne afin de pouvoir tester la compatibilité des chroniques qu'il reçoit.

Cette approche a été implémentée sous la forme d'un outil, baptisé CARDECRS[3] (Le Guillou *et al.*, 2008).

6.1. *Conséquences de la construction en ligne du modèle des interactions sur le calcul du diagnostic*

La possibilité d'utiliser les chroniques communicantes afin de diagnostiquer un système dépend de notre capacité, dans un contexte d'exécution donné, à retrouver les services ayant pris part à la composition, les instances ayant communiqué, et à fusionner correctement les variables qu'ils ont échangées. Cette section expose les problèmes posés par la fusion des chroniques communicantes, fortement liés à la disponibilité d'un modèle des interactions, que celui-ci soit construit en ligne ou non.

6.1.1. *Services et instances homologues*

Afin de tester la compatibilité de chroniques communicantes ou d'abrégés de chroniques, il est nécessaire de connaître l'adresse exacte des services ayant pris part à la composition considérée, et de savoir quelles instances des services sont liées dans cette instance de composition.

La section 4.3 introduit la notion de point de synchronisation comme étant un tuple $((inter_{id}, \{vars\}, serv_{id}), liste_{statuts})$ où $inter_{id}$ est l'identifiant du point d'interaction et donc du point de synchronisation associé, $\{vars\}$ une liste ordonnée de variables échangées, $serv_{id}$ un identifiant correspondant au service distant partenaire et $liste_{statuts}$ la liste des statuts associés à chacune des variables échangées.

À la création des chroniques communicantes, cet identifiant référence un appel à une opération extérieure, typiquement le nom d'une opération `invoke` en BPEL[4] :

3. Chroniques Appliquées à la Reconnaissance Distribuée d'Erreurs via CRS.
4. Dans le cas d'orchestrations ou de chorégraphies statiques, l'adresse de l'opération distante peut être fixée dans la chronique communicante.

`<invoke name="MonAppel" .../>`. À l'exécution, l'adresse de l'opération distante est fixée et le champ $serv_{id}$ des instances de chroniques communicantes en cours de reconnaissance est instancié. Les chroniques communicantes reconnues contiendront donc les informations de localisation des services partenaires, nécessaires à la fusion.

Quant aux instances de services appartenant à une même exécution, leur détermination dépend essentiellement des technologies utilisées pour implémenter le système distribué considéré. Dans le cadre le plus défavorable (composition dynamique de services sans mécanisme de support d'identifiant), chaque chronique communicante doit mémoriser, en plus de l'adresse de l'opération distante, le PID de l'instance de service invoquée. Dans le cadre le plus favorable (composition statique avec support d'un identifiant global), seul l'identifiant global de l'instance de composition doit être mémorisé. Dans les cas intermédiaires comme les nôtres (compositions statique ou dynamique sans identifiant global), certaines propriétés des langages utilisés peuvent résoudre notre problème, c'est le cas des ensembles de corrélation. Un ensemble de corrélation peut associer des messages à une instance de composition de services. En utilisant de tels messages pour identifier notre composition, il suffit aux chroniques communicantes de mémoriser, à l'exécution, l'identifiant contenu dans l'ensemble de corrélation pour être capables de fusionner correctement durant la phase de diagnostic global. Le fonctionnement des ensembles de corrélation n'est pas détaillé dans ce document.

6.1.2. *Correspondance des variables échangées*

Le point déterminant, dans la fusion de chroniques communicantes homologues, une fois les services et les instances d'exécution identifiés, réside dans l'identification des correspondances entre variables échangées par les différents composants. En effet, lorsque deux services échangent plusieurs messages au cours du même processus d'entreprise, les points de synchronisation associés doivent être identifiés avant fusion.

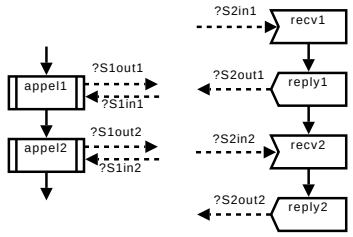


Figure 9. *Appels séquentiels et synchrones à un service distant*

Si certains cas, dans lesquels l'ordre des messages échangés entre les services est total (figure 9), se résolvent par une simple méthode consistant à lister les points de synchronisation dans leur ordre d'exécution, on peut imaginer des situations (à très faible probabilité) dans lesquelles deux branches concurrentes d'un service invoquent deux branches concurrentes d'un service extérieur (figure 10). Cette situation, bien que rare en pratique, nous montre qu'il est nécessaire de définir une méthodologie

claire concernant la définition des points de synchronisation. Dans ce but, la section 6.1.3 présente les exigences de la fusion de chroniques homologues dans le pire cas et définit une méthodologie permettant de satisfaire ces exigences. La section 6.1.4 illustre la méthode proposée.

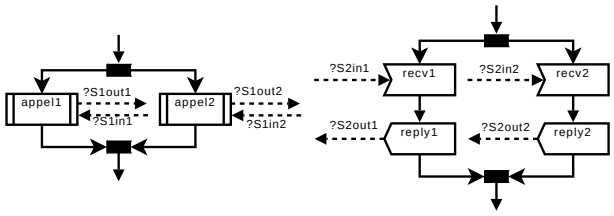


Figure 10. *Appels concurrentiels et synchrones à un service distant*

6.1.3. *Méthodologie de définition des points de synchronisation*

Le cas d'étude de la figure 10 met en évidence la nécessité d'identifier source et cible des points de synchronisation sous peine d'ambiguïtés lors de la fusion des chroniques communicantes homologues. En considérant cette composition de deux services, on identifie, sur chaque service, quatre points de synchronisation :

– appel de $appel1$ vers $recv1$;

– retour de $recv1$ vers $appel1$;

– appel de $appel2$ vers $recv2$;

– retour de $recv2$ vers $appel2$;

À partir de la définition de point de synchronisation donnée en section 4.3, on cherche alors à instancier le tuple $((inter_{id}, \{vars\}, serv_{id}), liste_{statuts})$ de manière à identifier de façon certaine les points de synchronisation homologues. Il n'est bien entendu pas possible de référencer le service distant dès l'écriture des chroniques communicantes, étant donné que les services sont supposés être composés dynamiquement. Nous proposons alors la méthode d'instanciation de tuple suivante.

6.1.3.1. Identifiant du point de synchronisation

En vue d'instancier l'identifiant du point de synchronisation $inter_{id}$, il est nécessaire de distinguer le fournisseur (service appelé) et le consommateur (service appelant) de l'opération.

Du côté fournisseur du service, l'identifiant à utiliser est le nom de l'opération appelée par le consommateur, tel qu'il est proposé au public. Du côté consommateur, le choix de l'identifiant est moins crucial et se portera plus facilement sur le type d'évènement supportant le point de synchronisation : nom de l'activité dans le cas d'un appel asynchrone, nom de l'activité suivi d'un marqueur de début ou de fin d'activité dans le cas d'un appel synchrone.

Dans l'exemple présenté en figure 10, on crée ainsi les identifieurs des points de synchronisation de chaque composant :

– $OpExterne1$ sur les points liés à $recv1$ et $reply1$, en admettant que l'adresse de l'opération soit de la forme `http://serviceB/OpExterne1` ;

– $OpExterne2$ sur les points liés à $recv2$ et $reply2$, en admettant que l'adresse de l'opération soit de la forme `http://serviceB/OpExterne2` ;

– $appel1Start$ et $appel1End$ sur les points liés à $appel1$;

– $appel2Start$ et $appel2End$ sur les points liés à $appel2$.

6.1.3.2. Ensemble de variables

Les variables précisées dans l'ensemble $vars_c$ correspondent aux paramètres d'appel ou aux valeurs de retour portés par la consommation du service distant. Dans un contexte mettant en jeu des langages dits classiques, l'ordre des variables respectera simplement l'ordre des paramètres d'appel de l'opération consommée. Dans un contexte mettant en jeu des langages échangeant une structure de données contenant les variables (dans un schéma XML, par exemple), la connaissance du schéma de la structure de données permettra d'identifier les variables homologues.

6.1.3.3. Identifiant de service distant

Contrairement à l'identification du point de synchronisation, l'identification du service distant ne requiert pas de distinction fondamentale entre le fournisseur et le consommateur de l'opération.

Pour le fournisseur de service comme pour le consommateur, un identifiant permettant de référencer l'opération de manière unique sera choisi. Cet identifiant sera instancié à l'exécution. Du côté fournisseur, il prendra pour valeur l'adresse du service appelant. Du côté consommateur, l'identifiant sera instancié avec la valeur effective de l'adresse de l'opération invoquée.

En reprenant l'exemple de la figure 10, on a les identifiants de services distants suivants :

– $clientOp1$ instancié à `http://serviceA/` sur les points liés à $recv1$ et $reply1$;

– $clientOp2$ instancié à `http://serviceA/` sur les points liés à $recv2$ et $reply2$;

– $appel1$ instancié à `http://serviceB/OpExterne1` sur les points liés à $appel1$;

– $appel2$ instancié à `http://serviceB/OpExterne2` sur les points liés à $appel2$.

6.1.4. *Illustration de fusion de chroniques communicantes dans le pire cas*

En respectant la méthodologie présentée en section précédente, on définit quatre points de synchronisation sur chacun des services de la figure 10. On a ainsi, pour le service fournisseur :

– $\mathcal{P}_{in_1}(f) = (OpExterne1, \{S2in1\}, clientOp1)$, point de synchronisation entrant ;

– $\mathcal{P}_{out_1}(f) = (OpExterne1, \{S2out1\}, clientOp1)$, point sortant ;

– $\mathcal{P}_{in_2}(f) = (OpExterne2, \{S2in2\}, clientOp2)$, point entrant ;

– $\mathcal{P}_{out_2}(f) = (OpExterne2, \{S2out2\}, clientOp2)$, point sortant.

De manière analogue, sur le service consommateur, on a :

– $\mathcal{P}_{out_1}(c) = (appel1Start, \{S1out1\}, appel1)$, point de synchronisation sortant ;

– $\mathcal{P}_{in_1}(c) = (appel1End, \{S1in1\}, appel1)$, point entrant ;

– $\mathcal{P}_{out_2}(c) = (appel2Start, \{S1out2\}, appel2)$, point sortant ;

– $\mathcal{P}_{in_2}(c) = (appel2End, \{S1in2\}, appel2)$, point entrant.

À l'exécution, la composition de services est instanciée et les services impliqués connaissent leurs partenaires. Les points de synchronisation sont à leur tour instanciés, et on a, pour le service fournisseur :

– $\mathcal{P}_{in_1}(f) = (OpExterne1, \{S2in1\}, \texttt{http://serviceA/})$;

– $\mathcal{P}_{out_1}(f) = (OpExterne1, \{S2out1\}, \texttt{http://serviceA/})$;

– $\mathcal{P}_{in_2}(f) = (OpExterne2, \{S2in2\}, \texttt{http://serviceA/})$;

– $\mathcal{P}_{out_2}(f) = (OpExterne2, \{S2out2\}, \texttt{http://serviceA/})$.

Sur le service consommateur, on a :

– $\mathcal{P}_{out_1}(c) = (appel1Start, \{S1out1\}, \texttt{http://serviceB/OpExterne1})$;

– $\mathcal{P}_{in_1}(c) = (appel1End, \{S1in1\}, \texttt{http://serviceB/OpExterne1})$;

– $\mathcal{P}_{out_2}(c) = (appel2Start, \{S1out2\}, \texttt{http://serviceB/OpExterne2})$;

– $\mathcal{P}_{in_2}(c) = (appel2End, \{S1in2\}, \texttt{http://serviceB/OpExterne2})$.

Ces points sont séparés, dans leurs chroniques respectives, selon leur direction : point de synchronisation entrant ou sortant. Ainsi, un point sortant défini sur le fournisseur ne peut fusionner qu'avec un point entrant sur le consommateur, et réciproquement. Le point entrant $\mathcal{P}_{in_1}(f)$ est donc potentiellement l'homologue d'un point sortant parmi $\mathcal{P}_{out_1}(c)$ et $\mathcal{P}_{out_2}(c)$, le point sortant $\mathcal{P}_{out_1}(f)$ étant lui-même l'homologue d'un point entrant parmi $\mathcal{P}_{in_1}(c)$ et $\mathcal{P}_{in_2}(c)$.

La discrimination du point homologue est effectuée grâce aux définitions des identifiants de point de synchronisation et de service distant données en section 6.1.3. Du côté fournisseur, l'identifiant de point de synchronisation est le nom de l'opération telle qu'elle est appelée par le client. Du côté consommateur, l'identifiant de service distant est instancié avec la valeur effective de l'adresse de l'opération invoquée. Un point de synchronisation possédant l'identifiant $opExt$ sur un service fournisseur sera donc l'homologue d'un point de synchronisation

dont l'identifiant de service distant est de la forme `http://adresse/opExt`. Il n'y a alors plus d'ambiguïté, et le point entrant $\mathcal{P}_{in_1}(f) = (OpExterne1, \{S2in1\}, \texttt{http://serviceA/})$ n'est compatible qu'avec le point sortant $\mathcal{P}_{out_1}(c) = (appel1Start, \{S1out1\}, \texttt{http://serviceB/OpExterne1})$. En procédant de même pour chaque point de synchronisation, on retrouve, sur la figure 10, les quatre homologies triviales de la figure 9 : $\mathcal{P}_{in_1}(f)$ et $\mathcal{P}_{out_1}(c)$, $\mathcal{P}_{out_1}(f)$ et $\mathcal{P}_{in_1}(c)$, $\mathcal{P}_{in_2}(f)$ et $\mathcal{P}_{out_2}(c)$, $\mathcal{P}_{out_2}(f)$ et $\mathcal{P}_{in_2}(c)$.

Cette étape d'identification des points de synchronisation homologues est une condition *sine qua non* de la fusion des chroniques communicantes.

6.2. *Répercussions sur l'algorithme de diagnostic*

6.2.1. *Construction d'une table de correspondance entre messages échangés*

En l'absence de modèle statique des interactions, il est indispensable de construire dynamiquement la table mettant en correspondance les variables homologues échangées par les différents services prenant part à une composition. En effet, comme l'expose la section 6.1.2, les variables homologues échangées par deux services n'ont aucune raison de porter le même nom, particulièrement en environnement dynamique. La construction de la table de correspondance vise alors à identifier ces variables homologues afin de pouvoir en comparer le statut dans les chroniques communicantes reçues, sans quoi tout diagnostic global est impossible.

En admettant que les chroniques communicantes aient été écrites en respectant la méthodologie exposée en section 6.1.3, considérons la table de correspondance suivante :

	Service source	Service distant	Var sources	Var distantes
1	$servA$	$servB$	$[?x, ?y]$	$[?z, ?t]$
2	$servB$	$servA$	$[?z, ?t]$	$[?x, ?y]$
3	$servA$	$servC$	$[?i, ?j]$	$[\,]$
4	$servC$	$servA$	$[\,]$	$[?i, ?j]$
...	...	...	...	...

Dans cet exemple, le diagnostiqueur global a reçu des informations de la part des serveurs $servA$ et $servB$, mais aucune information en provenance de $servC$. Il a alors été possible de remplir intégralement les deux premières lignes de la table, au contraire des deux lignes suivantes pour lesquelles il manque de l'information. À ce stade de la construction de la table, le diagnostiqueur global est capable de fusionner des chroniques en provenance de $servA$ et $servB$ en comparant le statut des variables $?x$ et $?z$, $?y$ et $?t$.

Les entrées 3 et 4 sont partielles car générées à partir des informations de $servA$ uniquement. À la réception du tuple $(servC, servA, \{?k, ?l\})$, ces lignes sont mises

à jour, mettant en correspondance les variables $\{?k, ?l\}$ de $servC$ avec les variables $\{?i, ?j\}$ de $servA$.

6.2.2. *Construction de l'arbre de diagnostic*

Le calcul du diagnostic global requiert la construction d'un arbre de diagnostic fusionnant les informations envoyées au diagnostiqueur global par les diagnostiqueurs locaux, que ce soit sur initiative d'un diagnostiqueur local (*push*) ou sur demande du diagnostiqueur global (*pull*). La figure 11 représente un exemple d'arbre de diagnostic.

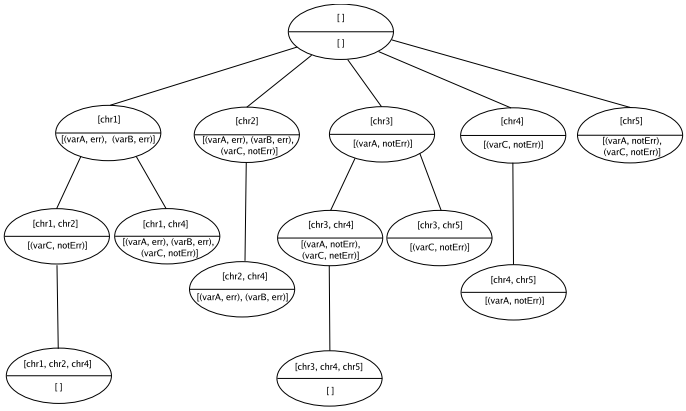


Figure 11. *Exemple d'arbre de diagnostic*

Comme l'illustre cet arbre de diagnostic, chaque nœud est composé de deux ensembles d'informations :

– un ensemble contenant les instances de chroniques communicantes ayant mené à l'étape de diagnostic global courante ;

– un ensemble de paires $(?variable, statut)$ contenant les contraintes de synchronisation restant à vérifier avant d'obtenir un diagnostic global.

L'ensemble de chroniques communicantes contient un nombre d'éléments égal à la profondeur du nœud dans l'arbre. En effet, chaque nœud possède l'ensemble des chroniques de son parent à laquelle on ajoute la chronique nouvellement reconnue.

La cardinalité de l'ensemble de contraintes de synchronisation, quant à elle, peut indifféremment augmenter ou diminuer. Elle augmente lorsque la chronique nouvellement reconnue n'élimine aucune contrainte de synchronisation déjà listée. À l'inverse, elle diminue lorsque le nombre de contraintes éliminées par la fusion de la chronique nouvellement reconnue est supérieur au nombre de contraintes ajoutées par cette même chronique. Aboutir à un nœud possédant un ensemble de contraintes vide signifie que toutes les contraintes liées à l'ensemble de chroniques associé à ce nœud sont vérifiées : on a alors un diagnostic global.

Dans l'exemple de la figure 11, il existe deux diagnostics globaux, ce qui se traduit par deux branches menant à un ensemble de contraintes vide :

– $\{chr1, chr2, chr4\}$

– $\{chr3, chr4, chr5\}$

Deux explications sont alors possibles quant à l'exécution globale de la composition. Soit les comportements modélisés par $chr1$, $chr2$ et $chr4$ se sont produits, soit ce sont les comportements modélisés par $chr3$, $chr4$ et $chr5$.

7. Construction de l'arbre global avec modèle statique des interactions

Dans cette section, nous nous situons dans le cas où la composition des services est statique et sa topologie connue a priori. Le modèle des interactions, utilisé par le diagnostiqueur global, est connu à l'avance. C'est cette connaissance a priori de la topologie de la composition de services qui nous permet, dans cette seconde approche, de nous appuyer, au niveau global comme au niveau local, sur des reconnaisseurs de chroniques.

Cette approche a été implémentée dans la plate-forme baptisée MATRAC[5] (Le Guillou *et al.*, 2009). Nous expliquons d'abord comment est extrait le modèle des interactions, puis comment le diagnostiqueur global tire parti de ce modèle lors du calcul du diagnostic.

7.1. *Extraction du modèle des interactions*

Nous supposons disposer d'une description des interactions entre services web partenaires dans un langage de description de compositions tel que WS-CDL. Cette description de la composition de services considérée est basée sur les structures d'ordonnancement et de contrôle classiques des langages impératifs.

Ainsi, à partir de la description des différents partenaires, des échanges entre partenaires et de la correspondance des messages portés par chaque échange, nous dérivons un modèle des interactions tel qu'il a été défini en section 4.2. Ce modèle nous permettra à la fois :

– de simplifier la fusion des chroniques locales en construisant avant l'exécution un équivalent à la table de correspondance entre variables homologues construite dynamiquement dans CARDECRS ;

– de dériver les points de synchronisation des chroniques communicantes locales à chaque service impliqué dans la composition (voir section 7.2.2) ;

5. Monitoring d'Architectures à Topologie Renseignée par Application de CRS

7.2. *Conséquences de la présence d'un modèle des interactions sur le calcul du diagnostic*

Le problème principal de l'approche sans modèle des interactions, baptisée CARDECRS, résidait dans le fait qu'il était impossible, a priori, d'établir une correspondance entre le nom d'un message émis par un service et le nom du même message reçu par le service partenaire. Pour pallier ce problème, une table de correspondance des variables homologues était générée dynamiquement (section 6.2.1).

Le modèle extrait de la description WS-CDL de la composition de services nous permet désormais de disposer de la correspondance entre variables et de générer hors ligne la table de correspondance. Dès lors, il nous est possible de générer les points de synchronisation des chroniques communicantes et des chroniques de fusion qui permettront le calcul du diagnostic global.

7.2.1. *Génération de la table de correspondance entre messages échangés*

La génération de la table de correspondance de variables homologues reprend intégralement l'ensemble des interactions entre partenaires. Cette table référence, pour chaque couple $(Service_A, Service_B)$, les variables portées par les échanges appartenant à la composition. Afin de faciliter la recherche de variables dans cette table, on agrège toutes les variables liées à un même couple de partenaires, puis on dédouble les informations de la table en produisant le couple inverse $(Service_B, Service_A)$.

La table de correspondance présentée en section 6.2.1, dans le cas de CARDECRS était construite dynamiquement. Cette fois-ci, le modèle des interactions nous fournit hors ligne toutes les informations nécessaires à sa construction. La table est donc complète avant même l'exécution de la composition.

7.2.2. *Génération des points de synchronisation des chroniques communicantes*

Tout comme il est courant, en partant de la description WS-CDL d'une composition, de dériver un squelette de chaque service en WS-BPEL, il est possible, à partir du modèle issu de cette description WS-CDL, de dériver un ensemble de squelettes de chroniques communicantes pour chaque service, chaque squelette contenant uniquement des points de synchronisation non instanciés.

Par exemple, sur le service Supplier, on peut générer un certain nombre de squelettes de chroniques communicantes, dont la chronique `SU::reserveFault` correspondant à une erreur de réservation d'un produit :

- $\mathcal{P}_{in_1} = ((ChkNReserve, (?SUlistIn), Shop), (?StSuListIn))$
- $\mathcal{P}_{out_1} = ((ReturnUnavail, (?SUlistOut), Shop), (?StSuListOut))$
- $\mathcal{P}_{out_2} = ((SendShipment, (?SUshipmentOut), Warehouse), (?StSuShipOut))$

Il ne reste alors plus, par connaissance experte, qu'à instancier les statuts associés aux messages portés par les points de synchronisation : $?StSuListIn = \neg err$, $?StSuListOut = \neg err$ et $?StSuShipOut = err$ dans cet exemple.

7.2.3. *Génération du fichier de chroniques de fusion*

La disponibilité a priori d'un modèle des interactions offre la possibilité, pour le diagnostiqueur global, de fusionner les diagnostics locaux et d'en déduire un diagnostic global selon deux conditions :

– l'écriture d'un abrégé de chaque chronique communicante pour les diagnostiqueurs locaux ;

– l'écriture de chroniques de fusion pour le diagnostiqueur global.

Grâce à ces éléments créés hors ligne et, par conséquent, intimement liés à la nature statique de la composition, nous pouvons concevoir un système reposant intégralement sur des reconnaisseurs de chroniques.

7.2.3.1. Abrégés de chroniques communicantes

Comme on l'a vu dans la section 5.1, un diagnostiqueur local, après avoir reconnu une chronique, envoie des informations assez précises quant au motif reconnu, résumées sous la forme d'un abrégé de chronique.

En considérant la chronique correspondant à une erreur de réservation sur le service Supplier présentée dans la section précédente, on génère le squelette d'abrégé de chronique (dont la notation a été quelque peu allégée ici)
`diagSupp[SUreserveFault,vert,?StSuListIn,?StSuListOut,?StSuShipOut]`
dont les statuts prendront pour valeur $\neg err$, $\neg err$ et err, comme en section 7.2.2.

7.2.3.2. Chroniques de fusion

Les chroniques de fusion du diagnostiqueur global reposent sur les évènements (abrégés) émis par les diagnostiqueurs locaux. Alors, pour chaque squelette d'abrégé généré sur un diagnostiqueur local, on génère un squelette plus générique sur le diagnostiqueur global, qui pourra reconnaître n'importe quel abrégé respectant ce format :
`diagSupp[?ChroSupp,*,?StSuListIn,?StSuListOut,?StSuShipOut]`

Une chronique de fusion est alors une chronique $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ où l'ensemble $\mathcal{S}$ contient un évènement émis par chaque diagnostiqueur local associé à un service prenant part à la composition, et $\mathcal{T}$ contient d'éventuelles contraintes temporelles sur ces évènements. En considérant la composition de services courante, une chronique de fusion peut se composer des trois abrégés de chroniques suivants :

```
diagShop[?ChroShop,*,?StShListSuOut,?StShListWHOut,?StShListSuIn,
                     ?StShListWHIn]
diagSupp[?ChroSupp,*,?StSuListIn,?StSuListOut,?StSuShipOut]
diagWH[?ChroWH,*,?StWHListIn,?StWHListOut,?StWHShipIn]
```

Des chroniques communicantes sont compatibles si et seulement si toutes les variables de synchronisation qu'elles partagent ont un statut identique. On définit alors une fonction `match` sur les statuts de variables. Grâce à cette fonction `match`, on gé-

nère automatiquement des contraintes sur les trois types d'évènements précédemment décrits en se servant de la table de correspondance des variables échangées par les services, ce qui nous conduit à la chronique de fusion suivante :

```
chronicle DiagGlobal {
  event(diagShop[?ChroShop,*,?StShListSuOut,?StShListWHOut,
                 ?StShListSuIn,?StShListWHIn],?t1)
  event(diagSupp[?ChroSupp,*,?StSuListIn,?StSuListOut,
                 ?StSuShipOut],?t2)
  event(diagWH[?ChroWH,*,?StWHListIn,?StWHListOut,
                 ?StWHShipIn],?t3)

  match[?StSuListIn,?StShListSuOut]
  match[?StSuListOut,?StShListSuIn]
  match[?StShListWHOut,?StWHListIn]
  match[?StShListWHIn,?StWHListOut]
  match[?StSuShipOut,?StWHShipIn]
}
```

À la reconnaissance de la chronique de fusion précédente, les variables $?ChroShop$, $?ChroSupp$ et $?ChroWH$ sont instanciées avec les noms des chroniques communicantes reconnues par les diagnostiqueurs locaux.

7.2.4. *Répercussions sur l'algorithme de diagnostic*

Comme l'a montré la section 7.2.3, le fait de disposer d'un modèle des interactions d'une composition de services nous permet, sur le diagnostiqueur global, de définir des "chroniques de fusion" capables de synchroniser des diagnostics locaux tout en vérifiant leur compatibilité grâce aux contraintes de synchronisation.

Ainsi, le diagnostiqueur global de MATRAC, contrairement à celui de CARDECRS, repose sur un reconnaisseur de chroniques gérant la construction de l'arbre de diagnostic, ce qui représente trois intérêts majeurs :

– simplifier au maximum le développement du diagnostiqueur global ;

– exprimer le diagnostiqueur global comme un diagnostiqueur local étendu, ce qui pourrait permettre de réaliser une hiérarchie de diagnostiqueurs ;

– réaliser la première plate-forme de diagnostic basée sur CRS entièrement distribuée.

8. Illustration sur l'exemple

Afin d'être fonctionnels, CARDECRS et MATRAC nécessitent une écriture hors ligne des modèles de diagnostic correspondant aux web services de la composition.

Pour chacune de ces approches, on doit fournir les modèles de chroniques communicantes et les abrégés associés. Ils sont élaborés à partir du workflow individuel de chaque web service (en respectant la méthodologie d'écriture pour CARDECRS). Pour MATRAC, on doit aussi fournir les chroniques de synchronisation extraites du fichier WS-CDL décrivant la composition.

Les plates-formes de diagnostic sont alors prêtes à surveiller la composition de services donnée. La suite de cette section présente le processus de diagnostic correspondant à l'exemple de *e-commerce* introduit dans la section 2. Nous nous intéressons en particulier à une exécution de cette composition dans laquelle une erreur de réservation (`SUreserveFault`) survient sur le fournisseur appelé Supplier (section 2.2) qui met alors de côté un produit ne correspondant pas à la commande initiale.

Le service Shop transmet aux services Supplier et Warehouse la confirmation de la commande du client. Le service Supplier prépare un colis contenant les produits du client (dont le produit erroné) et envoie ce colis au service Warehouse. Deux chroniques sont alors reconnues par le service Supplier, puisqu'elles correspondent à la même séquence d'évènements :

– `SU::normal`, correspondant à l'exécution normale du service ;

– `SU::reserveFault`, correspondant à une erreur de réservation non détectable par le service.

Ces deux chroniques étant classées *vertes* sont stockées dans le *tampon* du diagnostiqueur local du service Supplier.

Lorsqu'il reçoit le colis du service Supplier, le service Warehouse compare son contenu à la commande initiale du client et remarque une anomalie (`WHbadPackage`). Une note à l'attention du client est alors jointe au colis, puis le colis est complété par les produits réservés par le service Warehouse, et est envoyé au client. Une seule chronique est reconnue par le service Warehouse :

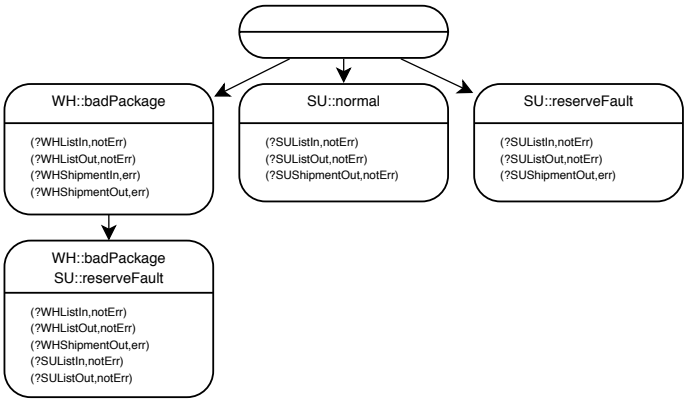
– `WH::badPackage`, correspondant à une non-conformité du colis reçu avec la liste des produits périssables commandés par le client.

Cette chronique est classée *rouge*, elle représente une erreur critique qui doit donner lieu immédiatement à un processus de diagnostic global. La chronique est donc envoyée au diagnostiqueur global qui crée un arbre de diagnostic contenant, sous la racine, un seul nœud correspondant à la chronique `WH::badPackage` dotée de ses informations de synchronisation.

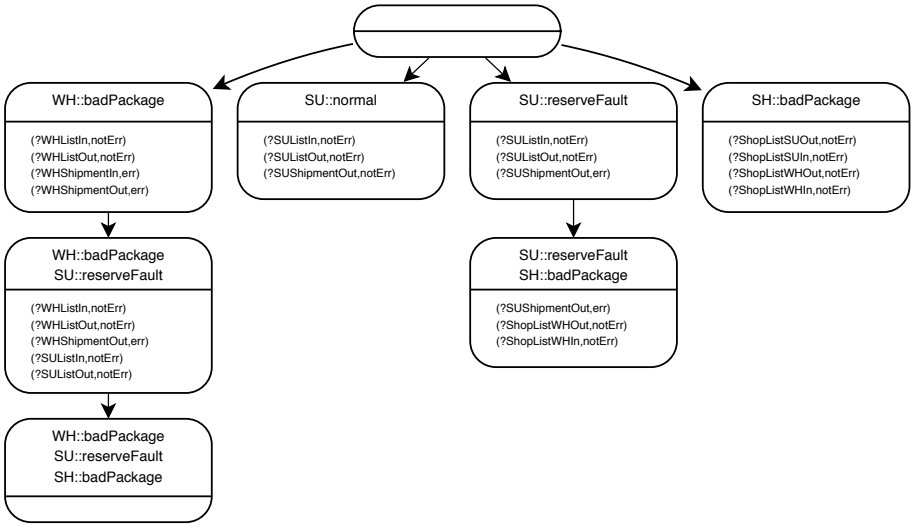
Le diagnostiqueur global prend alors l'initiative d'interroger les services partenaires du service Warehouse, à savoir les services Shop et Supplier, en leur demandant de purger leurs *tampons* et de laisser passer, désormais, les chroniques vertes.

Le service Shop n'a encore reconnu aucune chronique. Le service Supplier, par contre, a stocké les chroniques `SU::normal` et `SU::reserveFault`, et les envoie au diagnostiqueur global qui tente alors de les intégrer à son arbre de diagnostic. La chronique `SU::normal` n'est pas compatible avec la chronique `WH::badPackage` et

reste un nœud isolé sous la racine. La chronique `SU::reserveFault` est compatible avec `WH::badPackage` et est greffée sous le nœud correspondant. L'arbre de diagnostic est alors le suivant :



Enfin, le client reçoit le colis dans lequel le produit erroné a été remplacé par une note expliquant qu'un problème est survenu lors du traitement de la commande. Fort logiquement, le client exprime son mécontentement sur l'interface web du service Shop, ce qui a pour effet d'achever la reconnaissance de la chronique `SH::badPackage`. Cette chronique est directement envoyée au diagnostiqueur global qui la greffe dans son arbre de diagnostic.



La chronique `SH::badPackage` reconnue est compatible avec le nœud `(WH::badPackage,SU::reserveFault)` ; elle est greffée et donne naissance à un

nouveau nœud dont toutes les contraintes ont été vérifiées. L'incertitude initiale sur le service Supplier, pour lequel il nous était impossible de savoir si le processus s'était déroulé correctement ou non, est levée, et il est maintenant possible d'envisager une action de réparation en ayant connaissance de la panne et du service à la source du dysfonctionnement.

9. Protocole d'expérimentation et premiers résultats

Les deux approches présentées ci-avant, à savoir CARDECRS et MATRAC, ont été expérimentées dans le cadre du projet européen WS-DIAMOND conjointement à une seconde approche de diagnostic (Ardissono *et al.*, 2005), détaillée en section 10, selon le protocole suivant :

1) implémentation d'une composition de web services par un laboratoire non impliqué dans notre approche de diagnostic ;

2) saisie du modèle de diagnostic correspondant ;

3) greffe de nos plates-formes de diagnostic sur un moteur d'exécution de web services industriel (ActiveBPEL) ;

4) exécution de différents scénarii de la composition par un membre d'un autre laboratoire.

Les résultats ont montré la pertinence de nos approches sur les pannes observables d'un point de vue comportemental, sans jamais avoir à violer l'intimité des données utilisateur, étant donné que seule la trace d'exécution de chaque service est observée.

De plus, l'explosion potentielle du nombre d'instances de chroniques qui pouvait être à craindre est à relativiser compte tenu du mécanisme de synchronisation rendant le reconnaisseur de chroniques CRS utilisable sur des systèmes industriels importants.

Il reste toutefois à expérimenter la surveillance de compositions à proprement parler, consistant à pouvoir modifier en ligne le flot d'exécution des services. Tel est l'objet de nos recherches actuelles.

10. Travaux connexes et discussion

La surveillance en ligne de compositions de web services est actuellement considérée comme un problème aussi important que stimulant. Nous discutons, dans la suite, de travaux proches des nôtres, c'est-à-dire ceux s'intéressant aux pannes de type sémantique qui, en raison de leur propagation au travers des services, ne peuvent pas être traitées localement comme des exceptions et nécessitent un diagnostic.

Dans (Baresi *et al.*, 2005), les auteurs présentent une approche orientée surveillance dynamique. Des règles de surveillance sont exprimées sous la forme d'assertions WS-CoL, d'une manière explicite et externe. Ces règles sont ensuite liées au processus WS-BPEL sous la forme d'un proxy appelé *gestionnaire de surveillance*, au

moment du déploiement des services. Le principal avantage de cette définition externe est que le concepteur peut adapter le degré de contrôle à un contexte d'exécution spécifique sans avoir à reconstruire l'intégralité du processus d'entreprise. (Subramanian *et al.*, 2008) propose d'intégrer un moteur dédié "SelfHealBPEL" implémentant une politique d'auto-diagnostic et d'auto-réparation au moteur d'exécution de web services ActiveBPEL. Ce moteur dédié interagit avec le module de gestion d'ActiveBPEL et a pour rôle de surveiller, de diagnostiquer et, au besoin, de réparer le processus en cours d'exécution.

Dans ces deux cas, les mécanismes d'auto-diagnostic et d'auto-réparation sont partie intégrante du moteur BPEL : les moniteurs sont des services BPEL pouvant tourner sur des moteurs standard. Lorsqu'un problème est détecté, c'est-à-dire lorsqu'une propriété surveillée est détectée comme non vérifiée, il est alors plus aisé d'influer sur le processus BPEL en cours, par exemple en forçant sa terminaison ou en appliquant une stratégie de récupération sur erreur.

À l'inverse, (Barbon *et al.*, 2006) et (Mahbub *et al.*, 2005), qui effectuent également de la surveillance en ligne de web services BPEL, proposent d'utiliser des moteurs de surveillance externes, tournant parallèlement à l'exécution du processus BPEL. Dans (Mahbub *et al.*, 2005), les spécifications de surveillance sont exprimées comme des formules de calcul évènementiel. Dans (Barbon *et al.*, 2006), elles sont exprimées en langage RTML comme des formules de Past LTL. Ces deux approches sont moins intrusives que les deux précédentes et permettent de capturer des propriétés plus sophistiquées, via un langage de spécification à fort pouvoir d'expression, permettant par exemple d'exprimer des propriétés statistiques ou temporelles (Barbon *et al.*, 2006).

Nous proposons également de séparer clairement le processus BPEL de la tâche de surveillance. Dans notre cas, le modèle de surveillance est exprimé sous la forme d'un ensemble de chroniques et repose sur des évènements abstraits, tels que le début ou la fin d'une activité, ainsi que sur les messages échangés par les services.

(Ardissono *et al.*, 2005) propose une approche de surveillance à base de modèle proche de la nôtre. Un premier point commun est l'intérêt porté à la propagation des fautes entre services. De plus, leur architecture est très proche de celle que nous proposons, à savoir une approche décentralisée dans laquelle chaque web service est doté d'un diagnostiqueur local. Ces diagnostiqueurs locaux génèrent des hypothèses de diagnostic à partir d'un modèle local et d'observations, et déclenchent si nécessaire un diagnostiqueur global. Le diagnostiqueur global a pour rôle de fusionner les diagnostics locaux afin de calculer un diagnostic global en interrogeant les services de la composition et en vérifiant la cohérence des hypothèses locales. La principale différence est que les auteurs cherchent ici à expliquer, a posteriori, les alarmes détectées localement. En conséquence, le modèle qu'ils utilisent exprime les dépendances de données entre les entrées et les sorties de chaque activité. De manière similaire, les travaux de (Li *et al.*, 2009) s'appuient sur les dépendances entrées-sorties et proposent de les coder dans le cadre du formalisme des réseaux de Petri colorés ; les diagnos-

tics locaux et globaux sont calculés en tirant parti des algorithmes développés dans ce cadre.

À l'inverse, notre but est de surveiller, en ligne, les comportements normaux ou anormaux des services. Le modèle que nous utilisons exprime le workflow sous la forme d'évènements, correspondant aux débuts et fins d'activités, et de messages échangés.

Plus récemment, dans le cadre du projet WS-DIAMOND, les auteurs de (Ardissono *et al.*, 2005) ont proposé d'utiliser le modèle des interactions d'une composition décrite en JBPM. Ce modèle des interactions est exprimé sous la forme d'un réseau de Petri copiant la structure du workflow, et est utilisé pour surveiller le processus en analysant les messages échangés entre les services à un niveau global, ce qui permet d'apprécier le bon déroulement de la chorégraphie. On retrouve dans ces travaux la manière dont notre diagnostiqueur global exploite le modèle des interactions extrait de la description WS-CDL d'une chorégraphie.

Une autre approche a été proposée, mais à notre connaissance non poursuivie, par les auteurs de (Baazizi *et al.*, 2008), qui s'appuie sur une abstraction du code BPEL en un automate décrivant les interactions entre services. Les interactions effectives lors du traitement d'une requête sont stockées dans une base de données et un langage de requête de type SQL permet de vérifier certaines propriétés. Il est difficile cependant de voir comment fonctionne en ligne cette approche qui se prête plutôt à une analyse a posteriori.

Ce qui caractérise notre approche par rapport à ces travaux est la volonté de détecter et de diagnostiquer en ligne des pannes de type sémantique. Celles-ci se propagent en général dans les services interagissant avec le service où s'est produit la panne et nécessitent une analyse globale pour localiser le service responsable et permettre une action de réparation ou de reconfiguration qui assurera le maintien de la qualité de service.

11. Conclusion

Cet article a présenté une approche à base de modèles pour la surveillance de web services. Notre but est de tenir compte de la propagation des fautes et, ainsi, d'être capables de diagnostiquer le service sur lequel s'est produit la faute, même si le dysfonctionnement a été détecté plus tard, sur un autre service de la composition.

Nous proposons d'étendre l'approche par reconnaissance de chroniques, utilisée pour surveiller des systèmes industriels complexes, au contexte distribué des web services. Chaque web service est alors doté d'un diagnostiqueur local chargé d'analyser les traces d'exécution du service et de reconnaître, grâce au système de reconnaissance de chroniques CRS, les comportements décrits au préalable dans la base locale de chroniques. Un diagnostiqueur global est chargé de recevoir, grâce à un algorithme *push-pull*, les hypothèses de diagnostic établies par un diagnostiqueur local et de les

fusionner avec les hypothèses établies par les diagnostiqueurs locaux des services partenaires.

Dans cet article, nous présentons deux approches. Dans la première, nous supposons que l'on ne dispose pas de manière explicite du modèle des interactions, décrivant la manière dont les services interagissent entre eux pour répondre à la requête. Ce modèle est dynamique et ne peut qu'être construit en ligne, en cours d'exécution. Les contraintes de synchronisation, utilisées par le diagnostiqueur global pour fusionner les chroniques reconnues localement, doivent être construites en ligne, en fonction des messages échangés. Cette approche a été implémentée sous la forme d'une plate-forme, nommée CARDECRS.

Dans la seconde approche, nous supposons que la composition est statique et décrite dans le langage WS-CDL. Il est possible d'en extraire un modèle des interactions, et de construire automatiquement des chroniques de synchronisation. Le moteur de reconnaissance de chroniques CRS est alors utilisé par le diagnostiqueur global pour vérifier la cohérence globale des hypothèses locales de diagnostic et éliminer les hypothèses incohérentes. Une plate-forme de surveillance, nommée MATRAC, a été conçue pour implémenter cette approche.

Ces deux plates-formes ont été expérimentées dans un contexte de *e-commerce* dans le cadre du projet européen WS-DIAMOND.

Nos recherches portent actuellement sur deux axes principaux. Le premier consiste à imbriquer les tâches de diagnostic et de réparation de manière à réagir au mieux aux dysfonctionnements détectés. Ces travaux s'appuient sur la notion de *self-healability* étudiée dans le cadre du même projet européen (Cordier *et al.*, 2008). Le second consiste à faciliter la tâche d'acquisition des chroniques, actuellement effectuée de manière experte. L'idée générale serait de partir d'un modèle abstrait du code BPEL et de l'enrichir avec des informations relatives aux pannes du service et à leurs propagations, afin de construire les bases de chroniques locales de manière plus ou moins automatique.

12. Bibliographie

Alves A., Arkin A., Askary S., Barreto C., Bloch B., Curbera F., Ford M., Goland Y., Guizar A., Kartha N., Liu C. K., Khalaf R., Marin D. K. M., Mehta V., Thatte S., van der Rijn D., Yendluri P., Yiu A., Web Services Business Process Execution Language (WS-BPEL), Version 2.0, OASIS standard, Technical report, 2007.

Ardissono L., Console L., Goy A., Petrone G., Picardi C., Segnan M., Theseider Dupré D., « Enhancing Web Services with Diagnostic Capabilities », *ECOWS'05 (3rd European Conference on Web Services)*, p. 182–191, 2005.

Baazizi M., Sebahi S., Hacid M., Benbernou S., Papazoglou M., « Monitoring Web Services : A Database Approach », *in* LNCS (ed.), *ServiceWave 2008*, Springer, p. 98–109, 2008.

Barbon F., Traverso P., Pistore M., Trainotti M., « Run-Time Monitoring of Instances and Classes of Web Service Compositions », *ICWS'06 (IEEE International Conference on Web Services)*, p. 63–71, 2006.

Baresi L., Guinea S., « Towards dynamic monitoring of WS-BPEL processes », *International Conference on Service-Oriented Computing*, p. 269–282, 2005.

Ben Halima R., Drira K., Jmaiel M., « A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services », *ICWS'08 (IEEE International Conference on Web Services)*, p. 104–111, 2008.

Cordier M.-O., Pencolé Y., Travé-Massuyès L., Vidal T., « Characterizing and checking self-healibility », *ECAI'08 (18^{th} European Conference on Artificial Intelligence)*, p. 789–790, 2008.

Dousson C., Gaborit P., Ghallab M., « Situation recognition : representation and algorithms », *IJCAI'93 (International Joint Conference on Artificial Intelligence)*, p. 166–172, 1993.

Le Guillou X., Cordier M.-O., Robin S., Rozé L., « Chronicles for On-line Diagnosis of Distributed Systems », *ECAI'08 (18^{th} European Conference on Artificial Intelligence)*, p. 194–198, 2008.

Le Guillou X., Cordier M.-O., Robin S., Rozé L., « Monitoring WS-CDL-based choreographies of Web Services », *DX'09 (International Worshop On Principles of Diagnosis)*, p. 43–50, 2009.

Li Y., Ye L., Dague P., Melliti T., « A Decentralized Model-Based Diagnosis for BPEL Services », *ICTAI '09 : Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, IEEE Computer Society, Washington, DC, USA, p. 609–616, 2009.

Mahbub K., Spanoudakis G., « Run-time Monitoring of Requirements for Systems Composed of Web-Services : Initial Implementation and Evaluation Experience », *ICWS'05 (IEEE International Conference on Web Services)*, p. 257–265, 2005.

Papazoglou M., Georgakopoulos D., « Service-Oriented Computing : Introduction », *Communications of the ACM*, vol. 46, n° 10, p. 24–28, 2003.

Sampath M., Sengupta R., Lafortune S., Sinnamohideen K., Teneketzis D., « Diagnosability of Discrete Event System », *IEEE Transactions on Automatic Control*, vol. 40, n° 9, p. 1555–1575, 1995.

Subramanian S., Thiran P., Narendra N., Mostefaoui G., Maamar Z., « On the Enhancement of BPEL Engines for Self-Healing Composite Web Services », *SAINT'08 (International Symposium on Applications and the Internet)*, p. 33–39, 2008.

The WS-Diamond Team, « WS-Diamond : An Approach to Web Services - Monitoring and Diagnosis », *ERCIM News (European Research Consortium for Informatics and Mathematics)*, vol. 70, p. 25–26, july, 2007.

van der Aalst W., ter Hofstede A., Kiepuszewski B., Barros A., « Workflow Patterns », *Distrib. Parallel Databases*, vol. 14, n° 1, p. 5–51, 2003.