HAL
open science

An approach for Self-healing Transactional Composite Services

Rafael Enrique Angarita Arocha

To cite this version:

Rafael Enrique Angarita Arocha. An approach for Self-healing Transactional Composite Services. Other [cs.OH]. Université Paris Dauphine - Paris IX, 2015. English. NNT : 2015PA090051 . tel-01281384

HAL Id: tel-01281384

https://theses.hal.science/tel-01281384v1

Submitted on 2 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° attribué par la bibliothèque

Université Paris-Dauphine

ÉCOLE DOCTORALE DE DAUPHINE

THÈSE

préparée au **LAMSADE**

présentée par

Rafael ANGARITA

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS-DAUPHINE

Spécialité : INFORMATIQUE

An Approach for Self-healing Transactional Composite Services

Soutenue publiquement le 11 décembre 2015 devant le jury :

M.	Rukoz	Directeur de thèse	Université Paris-Dauphine, France Université Paris Ouest, France
M.	Manouvrier	Co-encadrant	Université Paris-Dauphine, France
S.	Reiff-Marganiec	Rapporteur	University of Leicester, U. K.
M.	Younas	Rapporteur	Oxford Brookes University, U. K.
D.	Grigori	Examinateur	Université Paris-Dauphine, France
S.	Dustdar	Examinateur	Vienna University of Technology, Austria
P.	Sens	Examinateur	Université Paris 6, France

Abstract

In this thesis, we present a self-healing approach for composite services supported by knowledge-based agents capable of making decisions at runtime. First, we introduce our formal definition of composite services, their execution processes, and their fault tolerance mechanisms using Colored Petri nets. We implement the following recovery mechanisms: backward recovery through compensation; forward recovery through service retry and service replacement; and checkpointing as an alternative strategy. We introduce the concept of Service Agents, which are software components in charge of component services and their fault tolerance execution control. We then extend our approach with self-healing capabilities. In this self-healing extension, Service Agents are knowledge-based agents; that is, they are self- and context-aware. To make decisions about the selection of recovery and proactive fault tolerance strategies, Service Agents make deductions based on the information they have about the whole composite service, about themselves, and about what is expected and what it is really happening at runtime. Finally, we illustrate our approach and evaluate it experimentally using a case study.

Keywords: Composite Service, Self-healing Systems, Fault Tolerance, Autonomic Computing, Dependability.

Résumé

Dans ce mémoire de thèse, nous présentons une approche d'exécution auto-corréctive (self-healing) de services composites, basée sur des agents capables de prendre, de manière autonome, des décisions pendant l'exécution des services, à partir de leurs connaissances. Dans un premier temps, nous définissons, de manière formelle, en utilisant des réseaux de Petri colorés, les services composites, leur processus d'exécution, et leurs mécanismes de tolérance aux pannes. Notre approche offre plusieurs mécanismes de reprise sur panne alternatifs : la récupération en arrière avec compensation ; la récupération en avant avec réexécution et/ou remplacement de service ; et le point de contrôle (checkpointing), à partir duquel il est possible de reprendre l'exécution du service ultérieurement. Dans notre approche, les services sont contrôlés par des agents, i.e. des composants dont le rôle est de s'assurer que l'exécution des services est tolérante aux pannes. Notre approche est également étendue afin de permettre un auto-recouvrement. Dans cette extension, les agents disposent d'une base de connaissances contenant à la fois des informations sur eux-mêmes et sur le contexte d'exécution. Pour prendre des décisions concernant la sélection des stratégies de récupération, les agents font des déductions en fonction des informations qu'ils ont sur l'ensemble du service composite, sur eux-mêmes, tout en prenant en compte également ce qui est attendu et ce qui se passe réellement lors de l'exécution. Finalement, nous illustrons notre approche par une évaluation expérimentale en utilisant un cas d'étude.

Mots clés : Services Composites, Système auto-corréctif, Tolérance aux Pannes, Informatique Autonome, Sûreté de Fonctionnement.

Acknowledgements

My PhD thesis has come to an end; however, it is far from being the result of only my work and effort. It has been possible thanks to numerous people who, in different ways, have had a positive influence during all these years. Unfortunately, I cannot list the name of every single person that I am grateful to.

Yudith Cardinale and the other Computer Science faculty of Universidad Simón Bolívar in Venezuela, without you, none of this would have even started.

My supervisors Marta Rukoz and Maude Manouvrier, for your continuous support and guidance throughout this thesis. Thank you for always been there for academic or non-academic support, and for always believing and trusting in me.

Stephan Reiff-Marganiec and Muhammad Younas, for all your support, for agreeing to review my thesis, and for the valuable feedback you gave me. Also, thank you for your patience despite all the bureaucratic problems we encountered. Schahram Dustdar, for all the feedback you gave me and for always being ready to answer my emails in the most sincere way. Daniela Grigori, for always being ready to help me and to talk with me despite your busy schedule. Pierre Sens, for immediately agreeing to be part of the jury of my defense.

The Computer Science faculty of LAMSADE, for always supporting me and trusting me. The administrative staff of LAMSADE, I would have been lost without you. The PhD students of Université Paris Dauphine, you made my PhD more interesting, fun, and enjoyable.

My family, for always being there for me, believing in my work, and my capacity, despite of living more than seven thousand kilometers away from me.

Laure, for dealing with my long nights working, unconventional work schedule, travels, and worries. Thank you for making my life better, supporting me, believing in me, and endless support and patience during all these years.

Résumé étendu

In this part, we present an extended resume of this thesis in French.

Chapitre 1. Introduction

Si nous regardons en arrière au début de la dernière décennie, plus précisément en 2001, quand tout était fait manuellement par des ingénieurs et des programmeurs, IBM a publié le manifeste *Autonomic computing : IBMs perspective on the state of information technology* [44] exprimant les préoccupations existantes au sujet de l'augmentation inévitable de la taille et la complexité des systèmes informatiques. Pour IBM, il était clair que cette complexité des systèmes hétérogènes et distribués minimiserait les avantages de la technologie de l'avenir ; par conséquent, la résolution du problème croissant de complexité était le "prochain Grand Challenge". Deux ans plus tard, est apparu l'article *The Vision of Autonomic Computing* [50] oú Kephart et Chess ont réaffirmé que la seule solution à la crise de la complexité du logiciel était à travers des systèmes informatiques qui s'auto-gérent. Ils ont présenté le concept de l'auto-gestion comme le bloc principal de la construction de l'informatique autonome. Le concept de l'auto-gestion est composé par les quatre aspects suivants : *auto-configuration*, *auto-optimisation*, *auto-guérison*, et *auto-protection*.

Dans cette thèse, nous nous concentrons sur la propriété d'auto-guérison, ce qui a été décrit par Kephart et Chess comme la capacité du système à détecter, diagnostiquer, et réparer automatiquement les pannes. Dans un article publié en 2007, Ghosh et ses coauteurs ont présenté les concepts maintenant bien connus concernant les propriétés et les états des systèmes auto-corréctifs [37]. Ils ont expliqué que la vision de systèmes à grande échelle était déjà une réalité et que la recherche sur les systèmes auto-corréctifs était active. En 2011, Psaiser et Dustar ont publié un article montrant le progrès de la recherche en auto-guérison [71]. Les

domaines de la recherche en auto-guérison concernent les systèmes embarqués, les systèmes d'exploitation, les systèmes basés sur l'architecture, les systèmes basés sur des multi-couches, les systèmes middleware-réflecteur, les applications multi-agents, la Programmation Orientée Aspect, les systèmes de découverte, et les systèmes basés sur les services Web et la Qualité de Service (QoS).

Plus récemment, le paradigme de l'Internet des Objet a gagné du terrain à la fois dans l'industrie et dans la recherche académique [17]. Il a également été inclus par le US National Intelligence Council dans le report "Disruptive Civil Technologies - Six Technologies With Potential Impacts on US Interests Out to 2025" [1]. L'Union Européenne a investi plus de 100 millions d'euros dans des projets liés à l'Internet des Objets, et le gouvernement de la Chine a publié le 12ème plan de développement de l'Internet des Objets [30]. Les applications de l'Internet des Objets sont censées avoir un impact énorme dans le domaine du transport et la logistique, de la santé, et des environnements intelligents. En effet, l'une des applications les plus importantes de l'Internet des Objets concerne le système de santé, lorsque les patients doivent être constamment surveillés par des dispositifs implantés qui communiquent automatiquement avec les systèmes hospitaliers [23, 88].

Dans *Internet of Things Strategic Research Roadmap* [91], Vermesan et ses coauteurs montrent que le comportement autonome et responsable des ressources est l'une des quatre tendances les plus importantes qui formeront l'avenir de l'Internet des Objets dans les prochaines années. Nous extrayons le paragraphe suivant en anglais :

> " ... the trend is towards the autonomous and responsible behaviour of resources. The ever growing complexity of systems, possibly including mobile devices, will be unmanageable, and will hamper the creation of new services and applications, unless the systems will show "self-*" functionality such as self-management, self-healing and self-configuration."

Même si cette thèse ne tient pas en compte des aspects spécifiques du paradigme de l'Internet des Objets, nous nous sommes inspirés du nombre croissant de services dû à l'explosion des objets connectés, et de l'impact que les applications composées par ces services auront sur nos vies. Par conséquent, dans cette thèse, nous nous concentrons sur les aspects d'auto-guérison de services composites, où les services composants peuvent être des services Web / API traditionnels, ou des services offerts par des objets dans l'Internet des Objets.

Contributions and publications

Cette thèse inclus les concepts et les résultats publiés dans les articles suivants :

[12] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. FaCETa: Backward and Forward Recovery for Execution of Transactional Composite WS. In *Proceedings of the Fifth International Workshop on REsource Discovery (RED 2012)*, pages 1–15, Heraklion, Grece, 2012

[77] Marta Rukoz, Yudith Cardinale, and Rafael Angarita. FACETA*: Checkpointing for Transactional Composite Web Service Execution based on Petri-Nets . *Procedia Computer Science*, 10(0):874 – 879, 2012

[26] Yudith Cardinale, Marta Rukoz, and Rafael Angarita. Modeling Snapshot of Composite WS Execution by Colored Petri Nets. In *Resource Discovery*, volume 8194 of *Lecture Notes in Computer Science*, pages 23–44. Springer Berlin Heidelberg, 2013

[14] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Reliable Composite Web Services Execution: Towards a Dynamic Recovery Decision . *Electronic Notes in Theoretical Computer Science*, 302(0):5 – 28, 2014

[13] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Dynamic Recovery Decision During Composite Web Services Execution. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, MEDES '13, pages 187–194, New York, NY, USA, 2013. ACM

[16] Rafael Angarita, Marta Rukoz, and Yudith Cardinale. Modeling dynamic recovery strategy for composite web services execution. *World Wide Web*, pages 1–21, 2015

[10] Rafael Angarita. Dynamic Composite Web Service Execution by Providing Fault-Tolerance and QoS Monitoring. In *Service-Oriented Computing - ICSOC 2014 Workshops and Satellite Events, Paris, France, November 3-6, 2014, Revised Selected Papers*, pages 371–377, 2014

[15] Rafael Angarita, Maude Manouvrier, and Marta Rukoz. A Framework for Transactional Service Selection Based on Crowdsourcing. In *Mobile Web and Intelligent Information Systems*, volume 9228 of *Lecture Notes in Computer Science*, pages 137–148. Springer International Publishing, 2015

[11] Rafael Angarita. Responsible Objects: Towards Self-Healing Internet of Things Applications. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 307–312, July 2015

Dans [12], nous avons proposé une approche pour l'exécution tolérante aux pannes des services composites qui étend celle de Cardinale et Rukoz [25]. Cette approche est basée sur la réexecution de service, le remplacement de service, et la compensation, et elle est formellement définie en utilisant le formalisme des réseaux Petri Colorés. Dans [77], nous avons proposé l'idée générale et le cadre d'un mécanisme de point de contrôle (*checkpointing*) pour les services composites. Ce mécanisme est une alternative au mécanisme de compensation proposé dans notre article précédent. Puis, nous avons présenté une modélisation formelle du mécanisme de checkpointing en utilisant les réseaux Petri Colorés [26].

Par la suite, nous avons présenté une approche de tolérance aux panne prenant en compte la Qualité de Service [14, 13, 16]. Dans [14], nous avons présenté une étude de l'impact des différentes stratégies de récupération sur les services composites. Dans [13], nous avons proposé un modèle pour décider dynamiquement de la stratégie de récupération en terme d'impact sur la QoS des services composites. Enfin, nous avons étendu le travail présenté dans les articles articles [14] et [13] pour formaliser un modèle de QoS pour la tolérance aux pannes de services composites [16]. Dans l'article [10], nous avons résumé la globalité de notre travail de recherche. Dans [15], nous avons présenté une approche de remplacement de service qui est complémentaire à notre système d'exécution pour la tolérance aux pannes de services composites.

Finalement, dans [11], nous avons présenté les perspectives futures que nous envisageons dans le domaine de l'auto-guérison des applications de l'Internet des Objets.

Notez que la plupart des titres de nos publications contiennent l'expression *service Web* ; cependant, aucune de nos approches ne dépend des technologies telles que SOAP ou styles architecturaux tels que REST. Nous abordons les services dans nos approches à un niveau plus conceptuel.

Chapitre 2. Notions préliminaires

Dans ce chapitre, nous introduisons quelques notions de base sur la tolérance aux pannes pour l'exécution des services composites. Nous illustrons l'Architecture

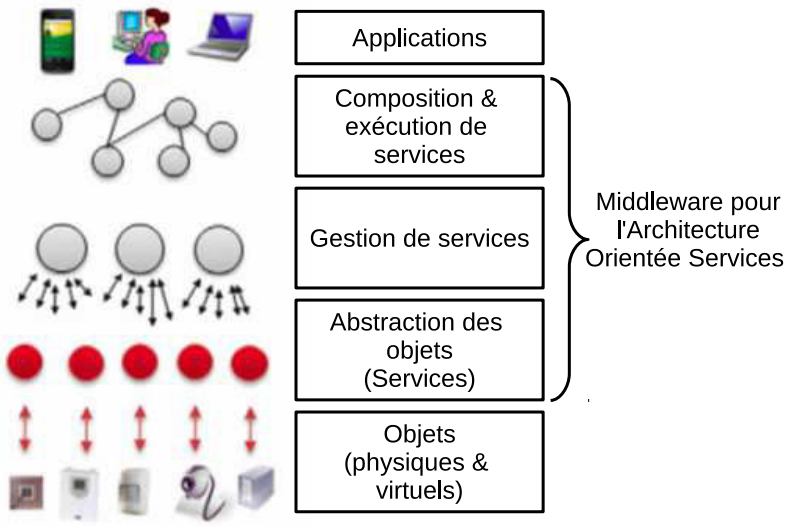


Figure 1: Middleware pour l'Architecture Orientée Services (modifiée de : Atzori et al. [17]).

Orientée Services et plaçons la contribution de cette thèse dans la couche Composition & exécution de services de la Figure 1. Ce middleware est composé des couches suivantes :

- Objets : les objets physiques sont situés dans cette couche. Quelques exemples d'objets physiques sont : les robots ; les téléphones mobiles ; les serveurs Web ; et les montres intelligentes. Nous considérons que les logiciels sans aucun lien avec le monde physique sont également dans cette couche.

- Abstraction des objets : dans cette couche, l'application et les objets logiciels sont représentés par un service publié sur Internet. En raison de la nature hétérogène des objets et des logiciels connectés, l'un des principaux objectifs de cette couche est d'harmoniser l'accès aux différents objets avec un langage et une procédure communs.

- Gestion de services : dans cette couche, les services publiés sont découverts et classés selon leur fonctionnalité. Par exemple, on peut trouver certaines approches pour la découverte de service dans [66] et [79].

- Composition & exécution de services : cette couche fournit la composition automatique des services en utilisant des approches telles que celle de [34].

Aussi, cette couche a en charge le suivi de l'exécution de ces services composites. Un aspect important de cette couche est la résilience aux pannes et l'adaptation en fonction des changements dans le système.

- Applications : les applications sont au-dessus de l'architecture SOA. Ces applications sont construites à partir des services et sont exécutées dans la couche Composition & exécution de services.

Nous situons la contribution de cette thèse dans la couche Composition & exécution de services du middleware SOA illustrée dans la Figure 1. Dans cette couche, les objets ont déjà été exposés en tant que services, ont été découverts, et classés par fonctionnalité. Les services composites sont créés automatiquement ou manuellement.

Comme nous l'avons expliqué, et comme son nom l'indique, le SOA tourne autour des services.

En 2004, le W3C a donné la définition de service *Web* suivante [92] :

> "A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided."

Cependant, cette définition a été donnée pour une norme particulière : les services *Web* SOAP (aussi connus comme *Big Web Services*) [76]. Avec SOAP, il existe un style architectural de développement des services : les services Web RESTful basés sur le paradigme *Representational State Transfer* (REST) [76]. Beaucoup d'ouvrages existent sur ces deux technologies, et il y a eu beaucoup de comparaisons des ces deux technologies afin de savoir laquelle est la mieux adaptée aux applications de services composites [67, 76]. Les tendances de la recherche et les services publiés sur Internet montrent que de nos jours les développeurs préfèrent les services RESTful aux services SOAP. Nous pouvons également remarquer que les services sont principalement décrits en langage naturel dans des pages Web ; par exemple, dans les annuaires de services publics tels que *ProgrammableWeb*[1]. En revanche, les annuaires publics les plus importants de

[1]Le répertoire de service le plus populaire : http://www.programmableweb.com

services SOAP (UDDI), maintenus par IBM et Microsoft, ont été fermés en 2006 (voir le Chapitre 10 de [76]). Les principales raisons de cette situation sont que : les technologies SOAP, WSDL et WS- * sont perçues comme complexes et ont rencontré de nombreux problèmes d'interopérabilité ; en revanche, les services dits RESTful sont légers et se basent sur des normes W3C/IETF telles que HTTP, XML, URI, MIME, l'infrastructure nécessaire est devenue omniprésente et il a été démontré qu'elle passe à l'échelle [67, 76].

Dans cette thèse, nous considérons une définition plus générale des services ; en d'autres termes, nous considérons que les services sont des opérations exposées sur Internet qui sont indépendantes de leur mise en œuvre. Par conséquent, les détails d'implémentations telles que SOAP ou REST sont hors du domaine d'investigation de cette thèse.

Les services sont décrits en fonction de leur fonctionnalité et des critères de qualité de service (QoS). Dans notre cas, la fonctionnalité d'un service est donnée par les paramètres d'entrée et de sortie. Nous supposons que les paramètres d'entrée et de sortie sont décrits en utilisant un langage d'ontologie telle que celle de [3]. En particulier, nous adoptons la relation d'ontologie *is-A*, que nous désignons comme $\subseteq_{is-A}$, pour déduire si un type de données est un *sous-type* d'un autre type de données. Par exemple, pour deux types de données d_1 et d_0, et la relation $d_1 \subseteq_{is-A} d_0$, nous disons que d_1 est un sous-type ou du même type que d_0. La Figure 2 illustre une ontologie composée de 11 types de données. Les arcs entre les types de données se réfèrent à la relation $\subseteq_{is-A}$, nous pouvons voir que tous les types de données sont des sous-types de d_0. Plus précisément :

$$
\begin{aligned}
&d_1 \subseteq_{is-A} d_5 \subseteq_{is-A} d_0\\
&d_{13} \subseteq_{is-A} d_2 \subseteq_{is-A} d_0\\
&d_6 \subseteq_{is-A} d_3 \subseteq_{is-A} d_{11} \subseteq_{is-A} d_0\\
&d_7 \subseteq_{is-A} d_4 \subseteq_{is-A} d_0\\
&d_8 \subseteq_{is-A} d_0
\end{aligned}
$$

En ce qui concerne la Qualité de Service, le standard ISO 9000 :2000 [47] définit la qualité comme :

> "The degree to which a set of inherent characteristics fulfills a need or expectation that is stated, general implied or obligatory."

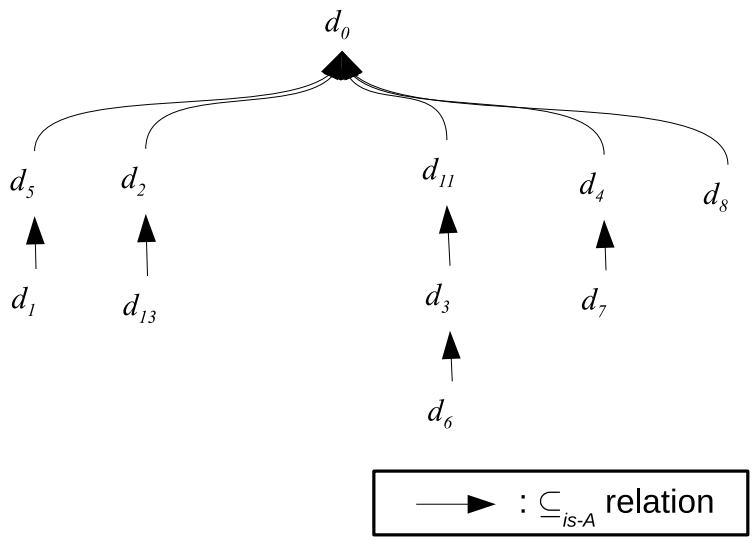


Figure 2: Ontologie d'exemple.

Il existe plusieurs modèles de qualité de service pour les services [73]. Dans ce qui suit, nous présentons notre définition des critères de Qualité de Service :

- *Temps de réponse* : le temps estimé nécessaire pour achever une invocation de service ; qui est, la durée entre une demande de service et la réponse du service correspondant.
- *Disponibilité* : la probabilité d'obtenir une réponse correcte après une invocation de service. Cela inclut la probabilité que le service est disponible, qu'il s'exécute correctement, et que la transmission de message entre le service et le demandeur a réussi.
- *Prix* : une mesure du coût d'exécution d'un service.

Parfois, l'opération exposée par un service élémentaire ne suffit pas pour résoudre une tâche particulière, donc nous avons besoin de combiner plusieurs services élémentaires. Ce nouveau service construit à partir de la combinaison de plus d'un service est appelé un *service composite*. Un service composite peut également être publié comme un service offrant une fonctionnalité et une QoS. Un scénario d'utilisation très utilisé par les chercheurs au cours de la dernière décennie pour étudier les compositions de service était l'application *Réservation de Voyage*[2]. Dans ce scénario, une société d'agence de voyage offre la possibilité de réserver

[2] http://www.w3.org/2002/04/17-ws-usecase.html

des forfaits de vacances complets, comprenant le transport, l'hébergement, les activités, etc. L'agence de voyage peut alors sélectionner les services qu'elle juge les plus appropriés pour créer un service composite qui répond aux demandes des ses clients. Par exemple, l'agence de voyage sélectionne les services fournissant une réservation d'avion, la location de voiture et la réservation d'hôtel. Notez que ces tâches sont interdépendantes, louer une voiture et réserver un vol peut être inutile si nous ne pouvons pas trouver un hôtel.

Pour résumer, nous disons qu'un service est une opération exposée sur le Web avec des propriétés fonctionnelles et non fonctionnelles ; de la même manière, un service composite est une combinaison de plus d'un service qui est également exposée sur le Web avec des propriétés fonctionnelles et non fonctionnelles.

Nous présentons les pannes que nous considérons dans cette thèse, le modèle transactionnel que nous utilisons pour fournir la tolérance aux panne automatique pour les services composites, et les principaux mécanismes de récupération : la récupération en arrière ; la récupération en avant ; et le checkpointing. Nous présentons également une introduction sur les systèmes auto-corréctifs et sur les Réseau de Petri colorés. Finalement, nous présentons un résumé des hypothèses générales de cette thèse.

Les propriétés transactionnelles les plus utilisées pour les services élémentaires sont *pivot*, *compensable*, et *retriable* [34]. Elles sont définies comme suit :

- Pivot(p) : un service est appelé pivot si ses effets restent pour toujours et ne peuvent pas être annulés sémantiquement une fois qu'il a terminé son exécution avec succès. Il s'agit de la propriété transactionnelle la plus basique.
- Compensable (c) : un service est compensable s'il existe un autre service qui peut sémantiquement annuler son exécution.
- Retriable (r) : un service est retriable s'il garantit une exécution réussie après un nombre fini d'invocations. Cette propriété doit être combinée avec les propriétés pivot ou compensable, créant les propriétés pivot-retriable (pr) et compensable-retriable (cr).

Les services composites construits à partir des services offrant des propriétés transactionnelles garantissent la cohérence du système et ont une propriété transactionnelle agrégée comme suit :

- Atomique (a) : un service composite est atomique si au moins un de ses services composant est pivot ou pivot retriable. Lorsqu'un service composite atomique se termine avec succès, ses effets demeurent pour toujours et ils ne peuvent pas être annulées. Si l'un de ses services composants tombe en panne, le système est laissé dans un état sémantiquement similaire à celui qu'il avait avant l'exécution du service composite.

- Compensable (c) : un service composite est compensable si tous ses services composants sont compensables. Cela signifie qu'il existe un autre service composite, contenant les services qui compensent les services du service composite compensable, qui peut annuler sémantiquement les effets du service composite compensable après son exécution réussie. Comme pour le service composite atomique, si l'un de ses service composants tombe en panne, le système est laissé dans un état sémantiquement similaire à celui qu'il avait avant l'exécution du service composite compensable.

- Retriable (r) : un service composite est retriable si tous ses services composants sont retriables. Un service composite retriable garantit l'exécution réussie après un laps de temps limité. Cette propriété doit être combinée avec les propriétés atomique ou compensable, pour créer les propriétés atomique-retriable (ar) et compensable-retriable (cr).

Les services qui fournissent des propriétés transactionnelles sont utiles pour créer des services composites fiables, assurant l'état cohérent de l'ensemble du système, même en présence des pannes. La reprise sur panne d'une exécution de service composite dépend de la propriété transactionnelle de ces composants : il faut alors utiliser des mécanismes de récupération en avant ou en arrière [25], ou retarder l'exécution du service composite [77]. Les principaux mécanismes de récupération sont les suivants :

- Récupération en arrière : elle consiste à restaurer l'état du système avant l'exécution du service composite ; c'est-à-dire, tous les effets produits par le service en panne sont annulées par *rollback*, et les effets des services exécutés avant la panne sont sémantiquement annulés en utilisant des techniques de compensation (voir Figure 3 (a)) ;

- Récupération en avant : elle consiste à réparer la panne pour permettre au service composite de poursuivre son exécution ; réessayer l'invocation de service ou trouver un service remplaçant sont des techniques utilisées pour fournir une récupération en avant (voir Figure 3 (b)).

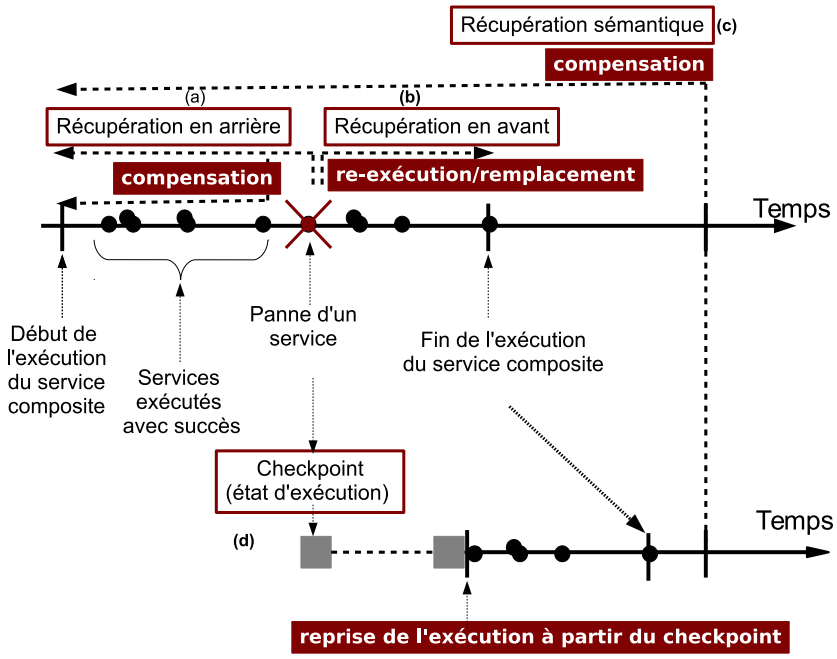


Figure 3: Techniques de récupération.

- Récupération sémantique : elle est similaire à la récupération en arrière, sauf que la récupération sémantique est effectuée après une exécution réussie d'un service composite en compensant l'exécution de ses services composants. L'idée est de laisser le système dans un état sémantiquement proche de l'état qu'il avait avant l'exécution du service composite (voir Figure 3 (c)).
- Checkpointing : si une panne survient, le checkpointing consiste en continuer l'exécution de la partie du service composite qui n'a pas été affecté par cette panne, tout en retardant l'exécution de la partie affectée (voir Figure 3 (d)).

Chapitre 3. Contrôle d'exécution de services composites et mécanismes de récupération

Dans ce chapitre, nous formalisons les services composites, leur exécution, et leurs mécanismes de tolérance aux pannes en utilisant les réseaux de Petri Colorés. Nous avons proposé un cadre composé de deux types de composants : un Coordinateur d'Agents responsable de la gestion des aspects globaux d'exécution

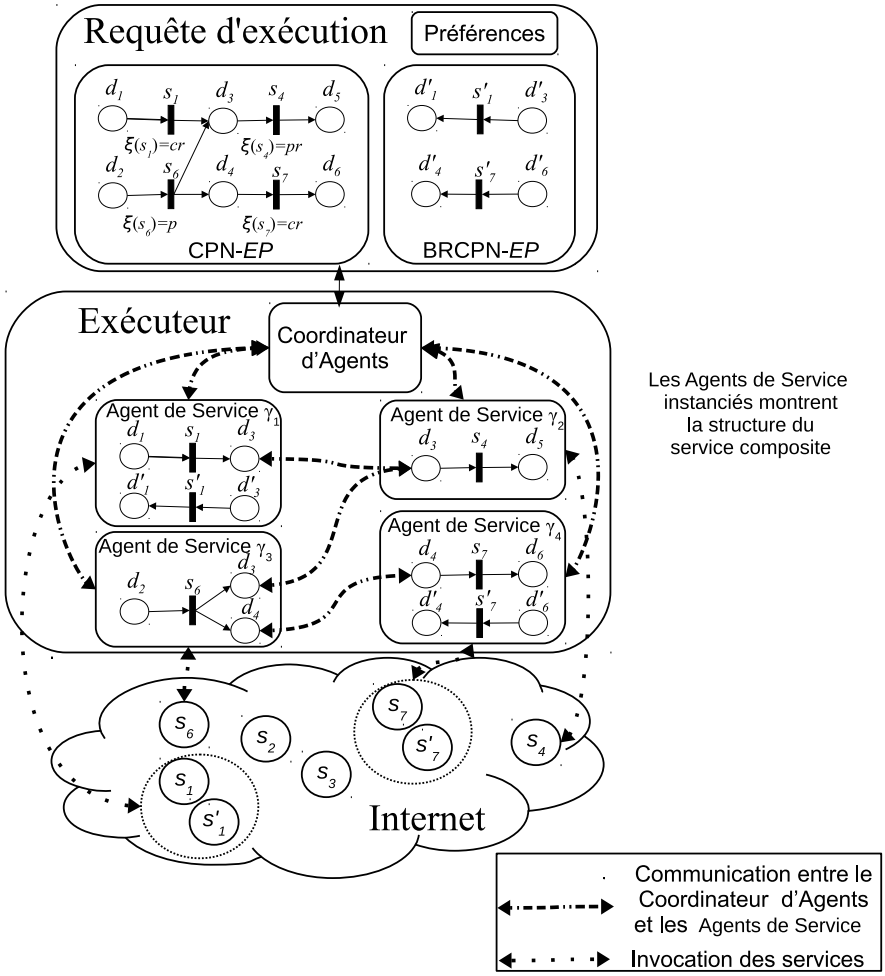


Figure 4: Architecture du système d'exécution.

de services composites ; et, les Agents de Service qui exécutent les services et sont en charge du contrôle de l'exécution et de la tolérance aux pannes. Notre cadre assure l'exécution correcte et tolérante aux pannes de services composites, et son modèle d'exécution distribué peut être implémenté dans des systèmes à mémoire distribuée ou partagée. Les mécanismes fournis par notre approche sont : la récupération en arrière par compensation, la récupération en avant par re-exécution de service et remplacement, la réplication, et le checkpointing.

Au cours de l'exécution du service composite, il existe deux variantes de base de scénarios d'exécution pour les services composants. Dans le scénario *séquentiel*, les services se basent sur les résultats des services précédents et ne peuvent être invoqués tant que les services précédents ne sont pas terminés. Dans le scénario *parallèle*, plusieurs services peuvent être invoqués simultanément, car ils n'ont pas

de dépendances de flux de données. La propriété transactionnelle globale d'un services composite est affectée par ces scénarios d'exécution. Par conséquent, il est obligatoire de suivre le flux d'exécution défini par le graphe du service composite pour s'assurer que l'exécution séquentielle et l'exécution parallèle satisfont la propriété transactionnelle globale.

L'exécution d'un service composite dans notre cadre est gérée par un Coordinateur d'Agents et une collection (Γ) d'Agents de Service (γ), organisés dans une architecture trois tiers. La Figure 4 représente l'ensemble de l'architecture de notre cadre. Dans le premier niveau, le Coordinateur d'Agents reçoit le service composite et son graphe de compensation correspondant, tous les deux représentés par des réseaux de Petri Colorés. Ces réseaux de Petri Colorés peuvent être générés automatiquement ou manuellement. Le Coordinateur d'Agent reçoit également la préférence indiquant si le mécanisme de checkpointing est activé ou non.

Le Coordinateur d'Agents lance dans la seconde couche un Agent de Service pour chaque service du service composite. La Figure 4 montre un exemple du cadre pour un service composite contenant les services :

$$S = \{s_1, s_4, s_6, s_7\}$$

Par conséquent, les Agents de Service suivants sont instanciés :

$$\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$$

où γ_1 est en charge de s_1, γ_2 de s_4, γ_3 de s_6, et γ_4 de s_7. Chaque agent de service est responsable du contrôle de l'exécution de son service ; c'est-à-dire, les Agents de Service :

- sont responsables de l'invocation de services ;
- surveillent l'exécution de leurs services correspondants ;
- envoient les résultats à leurs pairs selon le flux d'exécution ;
- prennent des actions de tolérance aux pannes en cas de panne.

En répartissant la responsabilité de l'exécution d'un service composite à travers de plusieurs Agents de Service, le modèle logique de notre exécuteur permet l'exécution distribuée et l'indépendance de la mise en œuvre. Par exemple, ce modèle peut être mis en œuvre dans un environnement de mémoire distribuée ou partagée.

L'idée est de placer le Coordinateur d'Agents et les Agents de Service dans différents noeuds physiques à haute disponibilité et fiabilité ; par exemple, dans un environnement de *cloud computing*. Les connaissances nécessaires à chaque Agent de Service peuvent être directement extraites des réseaux de Petri Colorés dans un environnement de mémoire partagée ou envoyé par le Coordinateur d'Agents dans une mise en œuvre distribuée.

Chapitre 4. Agents de services basés sur des connaissances

Dans ce chapitre, nous présentons les raisons pour lesquelles nous avons besoin de dynamisme pour la choix de la stratégie de tolérance aux pannes et de surveillance de la QoS : la récupération en avant est choisie si elle est possible ; si elle n'est pas possible, la récupération en arrière est choisie. Le *checkpointing* est sélectionné si l'utilisateur l'a choisi comme alternative à la récupération en arrière.

Avec le but de fournir un choix dynamique de la stratégie de tolérance aux pannes, dans ce chapitre, nous introduisons une approche auto-corrective pour les services composites. Dans cette approche, les agents de service sont des agents basés sur des connaissances. Ils font la sélection de la stratégie de tolérance aux pannes en se basant sur les informations qu'ils ont sur le service composite, sur eux-mêmes, et sur ce qui est attendu et ce qu'il se passe réellement pendant l'exécution. Sur cette base, notre conception considère une boucle d'auto-guérison par Agent de service pour effectuer la *détection*, le *diagnostic* et la *récupération* d'une manière décentralisée.

Le composant *détection* (Figure 5 (a)) prend en compte un source externe et deux sources de données internes. L'information externe concerne la QoS attendue ; par exemple, l'utilisateur peut permettre une certaine dégradation de la QoS. L'information interne se réfère à la dégradation de la QoS des services composants (par exemple, les variations négatives dans le temps d'exécution et

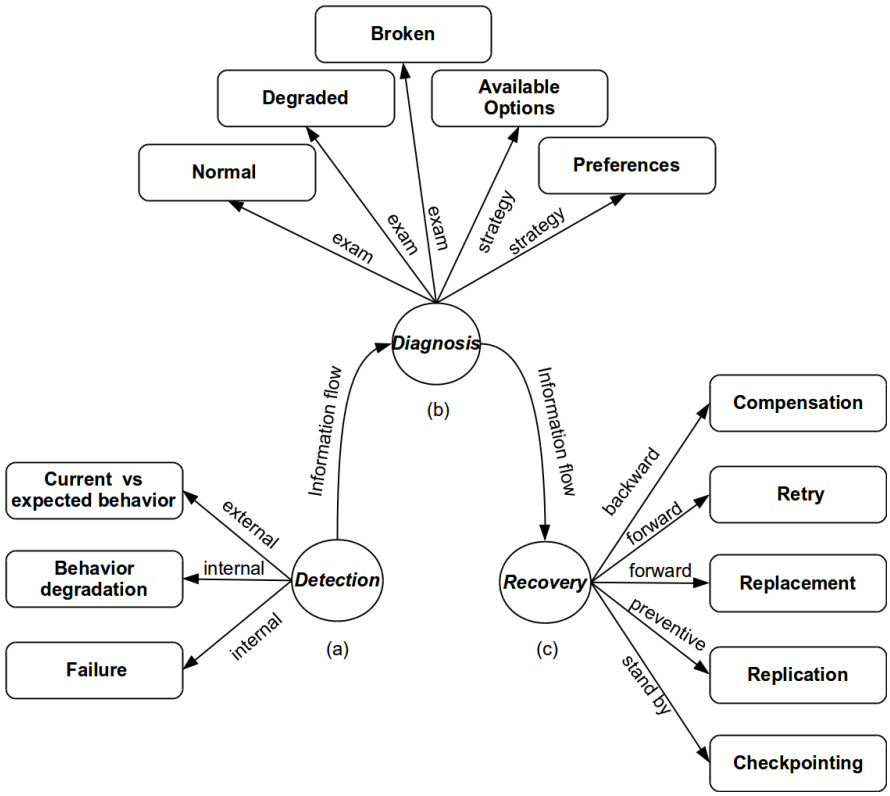


Figure 5: Boucle auto-corrective des Agents de Service.

le prix), et aux pannes de services, ce qui est également un cas particulier de dégradation de la QoS.

Le composant *diagnosis* (Figure 5 (b)) effectue l'analyse du problème et la détermination de l'état du service composite. Les trois diagnostics possibles correspondent aux trois états d'un système auto-correctif : *normal* ; *degraded* ; et *broken*. Le choix du mécanisme de récupération est influencé par les options disponibles (par exemple, les services de replacement disponibles, les propriétés transactionnelles, etc.), et les préférences de l'utilisateur (par exemple, la QoS attendue, le checkpointing, etc).

Le composant *recovery* (Figure 5 (c)) est en charge de l'exécution des mécanismes de tolérance aux pannes sélectionnés : la récupération vers l'arrière grâce à la compensation ; la récupération en avant par reexécution ou remplacement ; la prévention grâce à la réplication ; ou le retardement l'exécution par le checkpointing.

Chapitre 5. Évaluation expérimentale

Dans ce chapitre, nous présentons une mise en œuvre de notre cadre et une évaluation expérimentale en utilisant un cas d'étude. Pour ce cas d'étude, nous incluons une description de son scénario et de l'environnement correspondant. Nous nous sommes intéressés à l'observation du cas d'étude pour trois systèmes différents : un système sans tolérance aux pannes, un système transactionnel, et un système auto-correctif. Ces trois systèmes différents sont définis comme suit :

- *nt-sys* : il s'agit d'un système qui a pas de mécanismes de tolérance aux pannes. Si un service tombe en panne, le système génère une exception et arrête son exécution.
- *tp-sys* : il s'agit de notre système d'exécution transactionnel présenté dans le Chapitre 4. Il prend des décisions de récupération en tenant compte uniquement des propriétés transactionnelles des services composants.
- *sh-sys* : il s'agit de notre approche pour l'exécution auto-corrective du service composite présentée dans le Chapitre 5. Les décisions sont prises en utilisant l'information et les règles contenues dans les bases de connaissances des Agents de Service.

En conclusion, l'évaluation expérimentale présentée dans ce chapitre montre : (i), la nécessité de fournir des mécanismes de tolérance aux pannes pour les services composites ; (ii), comment notre approche du Chapitre 4 gère les pannes des services composites en utilisant les propriétés transactionnelles ; et (iii), comment notre approche auto-corrective du Chapitre 5 prend en compte la QoS pour la prise de décision. L'évaluation présentée dans ce chapitre suggère que la combinaison des propriétés transactionnelles avec des capacités d'auto-guérison conduit à des systèmes d'exécution plus intelligents ayant la capacité de gérer les exigences de haut niveaux pour les exécutions de services composites avec une intervention humaine minimale.

Chapitre 6. État de l'art

Dans ce chapitre, nous proposons une analyse des approches existantes pour l'exécution fiable de services composites. Ces approches peuvent être classées en

		Recovery mechanism											Eval.	
approach	*publication year*	*transactional properties*	*compensation*	*retry*	*substitution*	*replication*	*checkpointing*	*replaning*	*ignore*	*bepel*	*self-healing*	*intrusive*	*simulation*	*case study*
Wenan Tan et al. [89]	2015	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓
Zheng et al. [102]	2015	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓
Li et al. [54]	2014	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✓	✓
Bushehrian et al. [24]	2012	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Saboohi and Abdul [78]	2012	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
Behl et al. [20]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓	✓
Brzeziński and et al. [22]	2012	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Dillen et al. [32]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Abdeldjelil et al. [8]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Cardinale and Rukoz [25]	2011	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Liu et al. [55]	2010	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓
Simmonds et al.[84]	2010	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓
Sindrilaru et al. [85]	2010	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓
Zhou and Wang [104]	2010	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Lakhal et al. [51]	2009	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗
Yin et al. [96]	2009	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Moser et al. [61]	2008	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗	✓	✓
Subramanian et al. [86]	2008	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓
Halima et al. [40]	2008	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓
Moo-Mena et al. [60]	2008	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗
Gotze et al. [38]	2008	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Baresi and Guinea [19]	2007	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓
Modafferi and Conforti [59]	2006	✗	✓	✓	✗	✗	✗	✓	✗	✓	✓	✗	✗	✗
Merideth et al. [57]	2005	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓

Table 1: Travaux sélectionnés.

plusieurs catégories : la recherche sur la tolérance aux pannes et la recherche sur les systèmes auto-correctifs. Nous avons sélectionné les travaux les plus pertinents publiés entre les années 2005 et 2015, et nous les comparons avec l'approche proposée dans cette thèse. La Table 1 montre les travaux sélectionnés.

La plupart des auteurs évaluent leurs travaux respectifs en faisant des simulations d'exécutions de services composites en utilisant des cas d'études. Habituelle-

ment, les chercheurs se concentrent sur l'évaluation de la performance en mesurant la surcharge introduite par leurs approches et en la comparant avec des systèmes d'exécution de services composites sans mécanisme de tolérance aux pannes. En raison de l'absence d'un banc de test ouvert, réel, et accepté pour les exécutions de services composites, et de la complexité et des spécificités de la mise en œuvre de chaque approche, il est difficile de faire une bonne analyse comparative d'une manière quantitative. Comme les chercheurs des travaux que nous avons étudiés, nous nous limitons à la comparaison entre les approches par une analyse qualitative en décrivant les différentes approches en termes de ce qu'elles offrent.

Plusieurs des approches étudiées sont des propositions pour étendre la spécification SOAP et le langage WSDL ; par exemple, en interceptant, analysant et modifiant les messages SOAP échangés. Les services basés sur SOAP ont perdu leur popularité, et les registres publics les plus importants de services SOAP (UDDI) ont été fermés il y a longtemps en raison de leur faible adoption, tandis que la création de services RESTful semble être l'option préférée des développeurs des services actuels. Par conséquent, la recherche basée sur SOAP a perdu son impact, surtout lorsqu'elle se concentre sur des détails des enveloppes SOAP et des documents WSDL. Néanmoins,, leurs concepts principaux et les techniques mises en œuvre sont encore utiles indépendamment des technologies utilisées. Certains de ces concepts sont : les étapes de détection, la surveillance, et la récupération des systèmes auto-correctifs ; les techniques de redondance et de diversité de conception pour augmenter la disponibilité et la protection contre les pannes byzantines ; les techniques de roll-back et de compensation ; la médiation de données pour résoudre les incompatibilités des données ; les remplacement des services et des sous-graphes ; le checkpointing ; et les techniques de prédiction et d'optimisation de ces techniques. En outre, de nombreuses approches de tolérance aux pannes proposent des mécanismes de gestion des exceptions au niveau du langage. Nous estimons que la tolérance aux pannes doit être gérée à un niveau d'abstraction plus élevé. Enfin, certains travaux sont une combinaison entre des techniques pour les phases de conception et d'exécution. Bien que nous ne nions pas l'importance des techniques pour la phase de conception, dans cette thèse, nous nous sommes intéressés à ce qui se passe et au processus de décision pendant la phase d'exécution.

Dans cette thèse, tout d'abord, l'objectif est de formaliser les services composites, leur processus d'exécution, les mécanismes de tolérance aux pannes, et les capacités d'auto-guérison. Ce modèle formel est indépendant du langage et des technologies sous-jacentes. De plus, il nous semble pertinent d'avoir des agents intelligents capables de gérer un service pendant l'exécution d'un service compos-

ite. L'idée est d'avoir un seul type d'agent, appelé agent de service, chargé de l'exécution d'un service indépendamment de la technologie dans laquelle ce service a été développé. Les agents de service doivent être capables de prendre des décisions basées sur des connaissances du contexte et sur leurs propres connaissances. Parmi les stratégies de tolérance aux pannes que notre approche fournit, nous avons : la récupération en arrière par compensation, la récupération en avant par re-exécution de service et remplacement, la réplication, et le checkpointing. Enfin, nous proposons des concepts et des techniques qui peuvent être facilement mis en œuvre indépendamment des technologies sous-jacentes ; par exemple, les agents de service peuvent être des abstractions de services SOAP ou RESTful, et ces services peuvent représenter des objets physiques dans le monde réel. Ils peuvent également exposer les fonctionnalités de systèmes logiciels.

En comparaison avec les techniques de tolérance aux pannes mises en oeuvre dans les travaux de recherche du domaine, nous ne considérons pas les services vitaux et non vitaux. Des approches de remplacement de sous-graphes et de réorganisation du service composite, de tolérance aux pannes byzantine, et de la médiation de données peuvent être facilement ajoutées à notre approche.

Chapitre 7. Conclusions et perspectives

Dans cette thèse, nous avons proposé une approche auto corrective pour l'exécution de services composites. Notre approche est basée sur les propriétés transactionnelles et sur des agents à base de connaissances. Notre approche se situe dans la couche de Composition & exécution de services de l'Architecture Orientée Services présentée dans le Chapitre 2. Un des principaux avantages de notre travail est l'utilisation des propriétés transactionnelles comme notion de base pour la tolérance automatique aux pannes. Ainsi, nous avons mis en œuvre des mécanismes d'exécution et de récupération qui fonctionnent d'une manière automatique sans avoir besoin de développeurs ou de tout autre type d'intervention humaine. Puis, nous avons étendu notre approche transactionnelle avec des agents à base de connaissances capables d'analyser l'exécution d'un service composite, et de déduire de nouvelles informations à partir de cette analyse. L'information ainsi déduite joue un rôle crucial dans le processus de prise de décision lors de l'exécution. Un autre aspect important est le fait que notre approche étudie les services à un niveau conceptuel pour fournir un modèle formel pour les exécutions de services composites. Ce modèle est basé sur des concepts et techniques qui peuvent être facilement mis en œuvre indépendamment des technologies sous-

jacentes. Les services gérés peuvent être des services SOAP ou RESTful, avoir des équivalents physiques dans le monde réel, ou peuvent correspondre à des fonctionnalités de systèmes ou de ressources logicielles exposées sur Internet. Finalement, notre travail a également établi la base pour une recherche future intéressante sur la gestion et les aspects d'auto-guérison d'applications distribuées dans l'Internet du Futur. Nos principales contributions peuvent se résumer comme suit :

- Nous avons proposé une modélisation formelle des services composites, de leur processus d'exécution, et de leur mécanismes de tolérance aux pannes en utilisant les Réseaux de Petri colorés.
- Nous avons proposé une mise en œuvre en utilisant les mécanismes de tolérance aux pannes suivants : récupération en avant en réessayant ou en remplaçant un service, et récupération en arrière avec compensation.
- Nous avons introduit le mécanisme de checkpointing comme une alternative à la récupération en avant et à la récupération en arrière. En cas de panne, ce mécanisme permet l'exécution de la partie du service composite qui n'a pas été affecté par cette panne. La partie affectée peut rester en *stand-by* pour être exécutée après.
- Nous avons étendu notre approche tolérante au pannes avec des propriétés de systèmes auto-correctifs. Cette approche introduit des agents basés sur des connaissances et permet une prise de décision plus sophistiquée. Nous avons classé les types de connaissances des agents de la manière suivante : connaissances du contexte et auto-connaissance.

Limitations

Les limitations les plus importantes de cette thèse sont les suivantes :

1) le manque d'un banc de test accepté par la communauté scientifique pour faire des évaluations expérimentales.
2) la difficulté de déployer notre approche dans le monde réel en raison de l'absence d'automatisation et d'interopérabilité entre les services publiés par les entreprises.

Notez que ces limitations concernent tout le domaine de la recherche sur l'exécution du services composites et pas seulement cette thèse.

Perspectives

Nos perspectives de recherche concernent l'analyse des données générées par notre système d'exécution de services composites, la mise en place de mécanismes d'identification de pannes et de réaction plus sophistiqués, et la définition d'un cadre d'auto-guérison pour l'Internet des Objets.

L'exécution de services composites et le big data

Le nombre de services publiés sur l'Internet a augmenté depuis leur introduction ; de plus, l'apparition des objets connectés a fait que ce nombre de services augmente encore plus vite [91]. Des prédictions sur l'Internet des Objets estiment qu'il y aura plus de 16 milliards d'objets connectés d'ici à 2020 [87].

Une des conséquences de cette explosion d'objets connectés est la génération d'énormes quantités de données ; par conséquent, on prévoit que les volumes de messages échangés pourraient facilement atteindre entre 1000 et 10000 par personne par jour [91]. Des chercheurs reconnaissent que l'un des défis les plus importants est l'analyse de toutes les données générées par ces objets connectés, car ces données n'ont de valeur que si elles sont recueillies, analysées et interprétées [75].

Dans notre contexte, nos perspectives sont de collecter et de stocker les données générées par notre système d'exécution de services composites, y compris le comportement des services, ainsi que les stratégies et leur impact sur l'exécution du service composite. Ces données peuvent être analysées pour améliorer la sélection des services de remplacement et la prise de décisions de récupération et de stratégies proactives.

Identification des pannes

Dans notre approche, nous n'identifions pas le type de panne ; à la place, nous traitons les pannes des services de manière générale. En cas d'échec, nous appliquons les mécanismes de tolérance au pannes en fonction de la disponibilité de ces mécanismes et des préférences de l'utilisateur. Néanmoins, il est important d'identifier le type de pannes puisque différentes pannes peuvent exiger des réactions différentes [29]. Par exemple, une panne de délai d'attente peut être résolue par une nouvelle tentative d'exécution du service, tandis que d'autres pannes peuvent nécessiter un remplacement de service.

De même, nous avons supposé que les services sont gérés par des agents fiables qui ne tombent pas en panne. Par conséquent, il peut être pertinent de considérer les pannes des agents ; par exemple, lorsqu'un agent participant à l'exécution d'un service composite peut ne pas répondre.

Les systèmes auto-guérissants et l'Internet des Objets

Des chercheurs reconnaissent l'importance de la tolérance automatique aux pannes des services composites comme un élément essentiel dans le paradigme de l'Internet des Objets [68]. Comme nous l'avons expliqué dans cette thèse, les services ne sont plus seulement des opérations logicielles exposées sur l'Internet, mais aussi l'abstraction d'objets physiques capables de modifier le monde réel. Il est essentiel de reconnaître ce caractère émergent des services pour faire face aux nouveaux défis introduits par les services composites qui interagissent avec les deux mondes : le monde physique et le monde virtuel. Les pannes dans ce type de services composites peuvent conduire à la perte de temps de production, à des dommages matériels, des catastrophes environnementales, ou à la perte de vie humaine [9].

Mrissa et al. [63] introduisent le concept d'avatar comme une extension virtuelle pour les objets. Ces avatars ont un comportement autonome. Notre concept d'agent de service n'est pas loin de celui des avatars. Par conséquent, nous prévoyons d'étendre et d'adapter les agents de service avec des caractéristiques similaires à celles des avatars. Par exemple, au lieu d'instancier un nouvel agent de service pour chaque exécution d'un service composite, les agent de service peuvent rejoindre des applications pour collaborer et participer à la réalisation de l'objectif souhaité. Ainsi, nous obtenons un coordinateur de l'application à la place d'un coordinateur d'agents. L'idée est d'avoir un coordinateur d'application au lieu d'un coordinateur des agents pour les applications critiques. Une application critique est un service composite avec des exigences de disponibilité et de tolérance aux pannes élevées comme les applications pour la surveillance de santé [23], l'Industrie 4.0 [68], ou d'autres applications présentées dans [9]. Le coordinateur d'application gérera les agents de service participants, leur donnera les informations requises pour atteindre les objectifs de haut niveau, et gérera des mécanismes de déclenchement d'urgence si est nécessaire. En outre, il montrera l'état de santé de l'application, d'autres informations pertinentes, et fournira des paramètres d'administration à travers un site Web accessible aux utilisateurs.

Finalement, il est important de définir le sens des propriétés transactionnelles dans le cadre des applications de l'Internet des Objets. Des concepts tels que la

compensation, la ré-exécution, le remplacement et la réplication peuvent avoir des considérations particulières puisque nous considérons des services qui ont la capacité de changer le monde physique.

Contents

Table of Contents 1

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Research Question . . . 5
- 1.3 Challenges and Solution Requirements . . . 6
- 1.4 Contributions and Publications . . . 6
- 1.5 Organization . . . 8

2 Preliminaries 11
- 2.1 Service Oriented Architecture, Services, and Composite Services . 12
- 2.2 Composite Service Execution Control . . . 16
  - 2.2.1 Fault Hypothesis . . . 17
  - 2.2.2 Transactional Properties for Services . . . 18
  - 2.2.3 Recovery Mechanisms . . . 22
- 2.3 Self-healing systems . . . 24
- 2.4 Petri Nets and Colored Petri Nets . . . 26
- 2.5 Summary of General Assumptions . . . 30

3 Composite Service Execution Control and Recovery Mechanisms 33
- 3.1 Modeling composite service executions . . . 34
  - 3.1.1 Backward Recovery . . . 42
  - 3.1.2 Checkpointing . . . 48
  - 3.1.3 Service Replacement . . . 52
- 3.2 Framework Architecture . . . 57
  - 3.2.1 Fault Tolerance Algorithms . . . 60
- 3.3 Conclusions . . . 69

4 Knowledge-based Service Agents **71**
4.1 Motivation 72
4.2 A High-level Definition of Self-healing Composite Services 74
4.3 Knowledge-Based Service Agents 79
4.3.1 Self-awareness Knowledge 82
4.3.2 Context-awareness Knowledge 83
4.4 Knowledge Base 96
4.4.1 QoS State Deduction 99
4.4.2 Self-healing State Deduction 100
4.4.3 Action Deduction 101
4.5 QoS Manager for Summation/Product QoS Criteria 105
4.6 Algorithms 108
4.7 Conclusions 111

5 Experimental Evaluation **113**
5.1 Implementation Overview 114
5.2 Case Study 117
5.2.1 QoS dataset 118
5.2.2 The e-Health System 119
5.3 Results 125
5.3.1 Composite Service Behavior (*nt-sys*) 125
5.3.2 Experimental Comparison Between *nt-sys*, *tp-sys*, and *sh-sys* 128
5.3.3 Conclusions of Sections 5.3.2.1 and 5.3.2.2: *tp-sys* vs *sh-sys* 134
5.3.4 Self-healing Behavior 135
5.4 Summary of Experimental Evaluation 138

6 Fault tolerance and self-healing composite service execution: an state of the art **139**
6.1 Fault tolerance for composite services 140
6.1.1 Transactional Properties-based Approaches 141
6.1.2 Redundancy and Design Diversity-based Approaches . . . 142
6.1.3 Exception Handling-based Approaches 143
6.1.4 Prediction and Optimization Approaches 146
6.2 Self-healing execution of composite services 147
6.2.1 BPEL-based approaches 147
6.2.2 Non-BPEL-based approaches 148

6.3 Discussion 150

7 General Conclusions 153

7.1 Summary 153

7.2 Limitations 155

7.3 Future Research Directions 155

7.3.1 Fault Identification and Reaction 156

7.3.2 Self-healing Internet of Things Applications 156

7.3.3 Composite Service Execution and Big Data 157

A Algorithms 159

A.1 Expected Execution Time Knowledge and The Critical Path Method 159

A.1.1 Critical Path Example 160

A.2 Predecessors and Dependent Outputs 163

B Experiences on Random Composite Services 167

B.1 Estimated Execution Time and the Critical Path Algorithm . . . 168

B.2 Estimated Price and Availability 168

B.3 Dependent Outputs and Predecessors 169

bibliography 183

Chapter 1

Introduction

Contents

1.1 Motivation 3
1.2 Research Question 5
1.3 Challenges and Solution Requirements 6
1.4 Contributions and Publications 6
1.5 Organization 8

1.1 Motivation

If we look back to the beginning of the last decade, more precisely in the year 2001 when systems were manually built and managed by engineers and programmers, IBM published the *Autonomic Computing* [44] manifesto expressing their concerns about the inevitable increasing of the size and complexity of computer systems. For them, it was clear that such complexity of heterogeneous and distributed systems will minimize the benefits of future technology; therefore, solving the increasing complexity problem was the "next Grand Challenge". Two years later, we had the *Vision of Autonomic Computing* [50] where Kephart and Chess reaffirmed that the only solution to the software complexity crisis was through computing systems that can manage themselves. They presented the concept of self-management as the building block of autonomic computing. The self-management concept includes four main aspects: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*.

Kephart and Chess described the self-healing property of autonomic systems as the system's ability to automatically detect, diagnose, and repair software and hardware problems. In a survey published in the year 2007, Ghosh and his coauthors presented the now well-known concepts of self-healing states and properties [37]. They explained that the vision of large scale systems was already a reality and that self-healing research was active. By 2011, Psaiser and Dustar published a survey showing the advancements on self-healing research [71]. The collected self-healing research areas in [71] included embedded systems, operating systems, architecture based, cross/multi-layer-based, multi agent-based, reflective-middleware, legacy application and Aspect Oriented Programming, discovery systems, and Web services and QoS-based.

More recently, the Internet of Things paradigm has gained ground, both in the industry and in research worlds [17]. It was also included by the US National Intelligence Council in the "Disruptive Civil Technologies - Six Technologies With Potential Impacts on US Interests Out to 2025" conference report [1]. The European Union has invested more than 100 million euros in projects related to the Internet of Things, and the government of China released the 12th Five-Year Plan for Internet of Things development [30]. Failures in this type of applications may lead to loss of production time, equipment damage, environmental catastrophes, or loss of human life [9].

The world of things is much more dynamic, mobile, and failure prone than the world of computers, with contexts changing rapidly and in unpredictable ways [56]. In the *Internet of Things Strategic Research Roadmap* [91], Vermesan and his coauthors place autonomous and responsible behavior of resources as one of the fourth macro trends that will shape the future of the Internet of Things in the years to come. We extract the following paragraph:

> " ... the trend is towards the autonomous and responsible behaviour of resources. The ever growing complexity of systems, possibly including mobile devices, will be unmanageable, and will hamper the creation of new services and applications, unless the systems will show "self-*" functionality such as self-management, self-healing and self-configuration."

Services exporting functionalities of things may be accessed on the Web and may interact with existing traditional services to form value-added composite

services [42], which is one of the key principles of the Service-Oriented Computing [43]. During the execution of a composite service, different situations may cause a service failure [29]; due to the nature of services and their execution environment, they cannot be assumed to be stable. In this sense, more than ever before, the reliable execution of composite services becomes a key mechanism to cope with challenges of open-world applications in dynamic environments [82].

In this thesis, we are inspired by the growing number of services, the impact that applications composed by those services will have on our lives, and therefore, the increasing need of fault tolerance and self-healing mechanisms for composite services. We deal with services at a conceptual level; they may be traditional Web services/service APIs, or services offered by objects. Finally, the main objective of this thesis is to tackle the composite service reliability problem by: (i), modeling composite services, their execution processes, and fault tolerance mechanisms; and (ii), defining self-healing properties of composite services executions.

1.2 Research Question

> *Given that composite services are executed in dynamic, unpredictable, and heterogeneous environments, how may we provide QoS-aware fault tolerance for composite service executions?*

We tackle this question by formulating the following more specific questions:

1) If a service fails, which fault tolerance strategy is the most appropriate regarding QoS?

2) Even during failure free executions, is it possible to improve QoS by taking proactive measures?

By providing answers to these questions, our research aims to contribute to the field of Service Oriented Computing, specifically in the areas of fault tolerance and self-healing composite services.

1.3 Challenges and Solution Requirements

Some major challenges and solution requirements of conceiving a self-healing approach for composite services are the following:

- the formal modeling of composite services and their execution and fault tolerance processes;
- the QoS monitoring of the whole composite service execution;
- the guaranteeing of the system consistency even in the presence of failures by applying recovery mechanisms;
- the selection of the most appropriate recovery mechanism regarding QoS and user preferences;
- the system must be as autonomous as possible, functioning with minimal human intervention.

To deal with these challenges, we propose a self-healing composite service approach based on transactional properties and knowledge-based agents. We use transactional properties as a deep-seated notion for fault tolerance, and we implement knowledge-based agents that are capable of representing facts and gathering knowledge about a composite service execution to take smarter decisions at runtime.

1.4 Contributions and Publications

The plan we followed during this thesis is the following:

1) the study and modeling of fault tolerance mechanisms for transactional composite services;
2) the conception of a dynamic decision making mechanism for composite services executions;
3) the definition of self-healing capabilities for composite services.

In [12], we proposed a framework for the fault tolerant execution of composite services as a continuation of the work of Cardinale and Rukoz [25]. This approach is based on service retry, service replacement, and compensation, and it is formally defined using the Colored Petri net formalism. In [77], we proposed the general idea and framework of a checkpointing mechanism for composite services. This mechanism is an alternative to the compensation mechanism proposed in our previous paper. Then, we have presented the formal modeling of the checkpointing mechanism using Colored Petri nets in [26].

Later on, we presented a series of papers to provide QoS-aware fault tolerance for composite services [14, 13, 16]. In [14], we presented a study of the impact of the different recovery strategies on the execution time of composite services. In [13], we proposed a preliminary model to dynamically decide which recovery strategy is the best choice in terms of the impact on the composite service QoS. Finally, in [16] we presented a complete model defining the different types of knowledge required to make dynamic decisions during composite service executions. In the PhD Symposium paper [10], we summarized our work done so far and highlighted our main research question and contributions. In [15], we presented a service replacement approach which is complementary to our execution system for the composite services fault tolerance.

Finally, in [11] we presented some preliminaries ideas for building a framework to provide self-healing capabilities for Internet of Things applications. This framework serves mostly as future research directions of this thesis.

The complete list of published papers during this thesis is the following:

[12] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. FaCETa: Backward and Forward Recovery for Execution of Transactional Composite WS. In *Proceedings of the Fifth International Workshop on REsource Discovery (RED 2012)*, pages 1–15, Heraklion, Grece, 2012

[77] Marta Rukoz, Yudith Cardinale, and Rafael Angarita. FACETA*: Checkpointing for Transactional Composite Web Service Execution based on Petri-Nets . *Procedia Computer Science*, 10(0):874 – 879, 2012

[26] Yudith Cardinale, Marta Rukoz, and Rafael Angarita. Modeling Snapshot of Composite WS Execution by Colored Petri Nets. In *Resource Discovery*, volume 8194 of *Lecture Notes in Computer Science*, pages 23–44. Springer Berlin Heidelberg, 2013

[14] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Reliable Composite Web Services Execution: Towards a Dynamic Recovery Decision . *Electronic Notes in Theoretical Computer Science*, 302(0):5 – 28, 2014

[13] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Dynamic Recovery Decision During Composite Web Services Execution. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, MEDES '13, pages 187–194, New York, NY, USA, 2013. ACM

[16] Rafael Angarita, Marta Rukoz, and Yudith Cardinale. Modeling dynamic recovery strategy for composite web services execution. *World Wide Web*, pages 1–21, 2015

[10] Rafael Angarita. Dynamic Composite Web Service Execution by Providing Fault-Tolerance and QoS Monitoring. In *Service-Oriented Computing - ICSOC 2014 Workshops and Satellite Events, Paris, France, November 3-6, 2014, Revised Selected Papers*, pages 371–377, 2014

[15] Rafael Angarita, Maude Manouvrier, and Marta Rukoz. A Framework for Transactional Service Selection Based on Crowdsourcing. In *Mobile Web and Intelligent Information Systems*, volume 9228 of *Lecture Notes in Computer Science*, pages 137–148. Springer International Publishing, 2015

[11] Rafael Angarita. Responsible Objects: Towards Self-Healing Internet of Things Applications. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 307–312, July 2015

Note that most of these publications contain the phrase *Web service*; however, none of them depends on technologies such as SOAP or architectural styles such as REST. They deal with services at a conceptual level.

1.5 Organization

We structure the rest of this document as follows:

- Chapter 2 introduces some basic notions about fault tolerant execution of composite services. These notions are: what a service and a composite service are, and their place in the Service Oriented Architecture; a fault

hypothesis, transactional properties, and recovery mechanisms for composite services; an introduction to self-healing systems and Colored Petri nets; and finally, a summary of the general assumptions of this thesis.

- Chapter 3 presents our approach to reliable execution of composite services based on transactional properties. We formalize the execution and the fault tolerance execution control of composite services using Colored Petri nets. We consider the following recovery techniques: backward recovery through compensation; forward recover through retrying and replacement; and checkpointing as an alternative stand-by strategy.

- Chapter 4 builds on top of the framework presented in Chapter 3 to provide self-healing composite service executions. The self-healing approaches still uses transactionality as a deep-seating notion of system consistency; however, it employs knowledge-based Service Agents to make recovery and preventive decisions considering the composite service execution context.

- Chapter 5 presents an implementation of our framework and evaluate it experimentally using a case study. We implement and compare three different systems: a non-fault tolerant system, the transactional approach of Chapter 3, and the self-healing approach of Chapter 4.

- Chapter 6 proposes a review of existing approaches to support the reliable execution of composite services. This review includes approaches that may be classified as fault tolerance and self-healing research.

- Chapter 7 concludes this thesis with a summary, contributions, limitations, and future research directions.

Chapter 2

Preliminaries

Contents

2.1 Service Oriented Architecture, Services, and Composite Services 12

2.2 Composite Service Execution Control 16

2.2.1 Fault Hypothesis 17

2.2.2 Transactional Properties for Services 18

2.2.3 Recovery Mechanisms 22

2.3 Self-healing systems 24

2.4 Petri Nets and Colored Petri Nets 26

2.5 Summary of General Assumptions 30

This chapter recalls some important concepts and the context used throughout this thesis. Section 2.1 presents the concepts of service and composite service, and their place in the Service Oriented Architecture. Section 2.2 presents the concepts related to the execution control of composite services which include: a fault hypothesis to answer the question of which faults our system tolerates and which it does not tolerate; the transactional model for composite services in which we base part of our work; and the existing recovery mechanisms for composite services using transactional properties. Section 2.3 provides a brief background on self-healing systems. Section 2.4 presents an introduction to the Petri net formalism we use to model composite services, their execution processes, and fault tolerance mechanisms. Finally, Section 2.5 summarizes the main assumptions that we consider in this thesis.

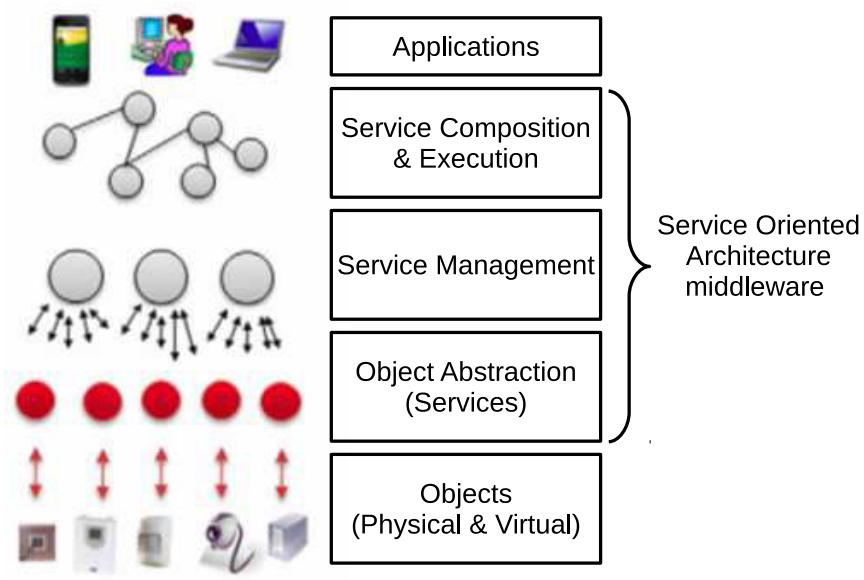


Figure 2.1: Service Oriented Architecture based middleware (modified from Atzori et al. [17]).

2.1 Service Oriented Architecture, Services, and Composite Services

The Service Oriented Architecture (SOA) [4] provides a general architecture for building service-based applications. Figure 2.1 illustrates a SOA based middleware inspired from the one presented by Atzori and his coauthors in their 2010 Internet of Things survey [17]. This middleware is composed by the following layers:

- Objects: physical objects are located in this layer. Some examples of physical objects are: robots; mobile phones; Web servers; and smart watches. We consider that software with no connection with the physical world is also in this layer.
- Object Abstraction: in this layer, software applications and objects are represented by services published on the Internet. Given the heterogeneous nature of connected objects and software, one of the main goals of this layer is to harmonize the access to the different objects with a common language and procedure.

- Service Management: in this layer, published services are discovered and classified according to their functionalities [58].
- Service Composition & Execution: it provides automatic service composition approaches [33, 74]. This layer is also in charge of the execution control of these composite services. One important aspect of this layer is the resilience to failures and the adaptation according to changes in the system.
- Applications: final user applications are on top of the SOA architecture. These applications are built from services and executed in the Service Composition & Execution layer.

We situate the contribution of this thesis in composite service execution control of the Service Composition & Execution layer of the SOA middleware depicted in Figure 2.1. In this layer, objects have already been abstracted and exposed as services, discovered, and classified by functionality, and composite services may be created either automatically or manually.

As we have explained, and as its name indicates, SOA revolves around services. In 2004, the W3C gave the following definition of *Web* service [92]:

> "A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided."

However, this definition was given for a particular standard: SOAP *Web* services (also known as Big *Web* Services) [76]. Along with SOAP, there exists an architectural style for developing services: RESTful Web services based on the Representational State Transfer (REST) paradigm [76]. There are already whole books about these two technologies, and there have been a lot of discussions about which one of these two technologies is the best or better suited for composed applications [67, 76]. Research trends and published services in the Internet show that nowadays developers prefer to build RESTful services over SOAP services. Also, we will notice that services are currently mostly described via narrative Web pages in natural language; for example, in public service directories such as

ProgrammableWeb[1]. In contrast, the most important public directories of SOAP services (UDDI), maintained by IBM and Microsoft, were shut down in 2006 (see Chapter 10 of [76]). Some reasons why this has happened are: SOAP, WSDL, and the WS-* stack are perceived as complex and have encountered many interoperability problems; in contrast, RESTful services are lightweight, they leverage on existing well-known W3C/IETF standards (HTTP, XML, URI, MIME), and the necessary infrastructure has already become pervasive and scalable [67, 76].

In this thesis, we consider a conceptual, more general definition of services; in other words, we consider that services are exposed operations on the Internet which are independent of their implementation. Hence, we do not dig into the details of service implementations such as SOAP or RESTful.

Services are described according to their functionality and Quality of Service (QoS) criteria. In our case, the functionality of a service is given by the input attributes it needs to be invoked, and the output attributes it produces after a successful invocation. We suppose that service inputs and outputs attributes are described using an ontology language, such as the Web Ontology Language [3]. In particular, we adopt the *is-A* ontology relation, which we denote as $\subseteq_{is-A}$, to deduce if a data type is a *subtype* of another data type. For example, given two data types d_1 and d_0, and the relation $d_1 \subseteq_{is-A} d_0$, we say that d_1 is a subtype or the same type as d_0. Figure 2.2 illustrates an ontology composed of 11 data types. Arcs between data types refer to the $\subseteq_{is-A}$ relation, from where we can see that all the data types are subtypes of d_0. More specifically:

$$
\begin{aligned}
&d_1 \subseteq_{is-A} d_5 \subseteq_{is-A} d_0 \\
&d_{13} \subseteq_{is-A} d_2 \subseteq_{is-A} d_0 \\
&d_6 \subseteq_{is-A} d_3 \subseteq_{is-A} d_{11} \subseteq_{is-A} d_0 \\
&d_7 \subseteq_{is-A} d_4 \subseteq_{is-A} d_0 \\
&d_8 \subseteq_{is-A} d_0
\end{aligned}
$$

Regarding service QoS, the standard ISO 9000:2000 [47] defines quality as:

> "The degree to which a set of inherent characteristics fulfills a need or expectation that is stated, general implied or obligatory."

[1]The most popular service directory: http://www.programmableweb.com

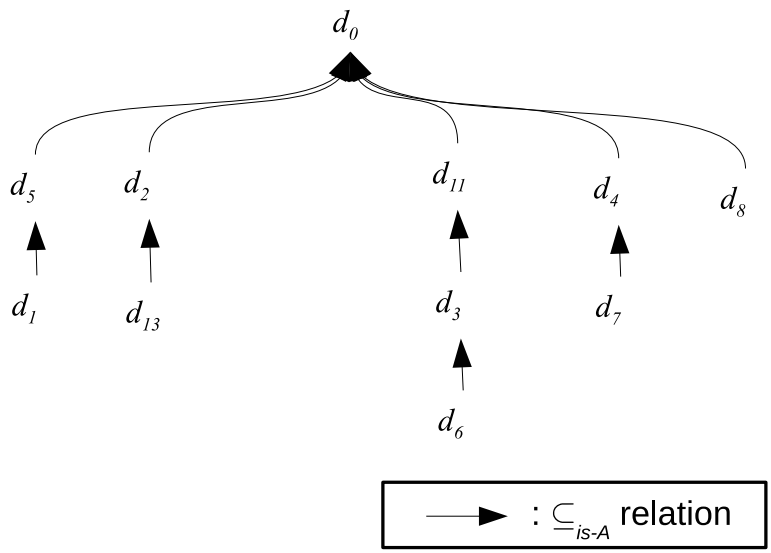


Figure 2.2: An Example Ontology.

There exist several QoS models for services [73]. In the following, we present our definition of service QoS:

- *Execution time*: the estimated time required to complete a service invocation; that is, the estimated time between making a service request and receiving its corresponding service response.
- *Availability*: the probability of getting a correct answer after a service invocation. This includes the probability that the service is up and it executes successfully, and that the message transmission between the service and the requester is successful.
- *Price*: a measure of the cost of invoking a service.

Sometimes, the operation exposed by a single elementary service is not enough to solve a particular task, so we require the combination of several elementary services. This new service built from the combination of more than one service is called a *composite service*. A composite service may also be published as a service providing a functionality and QoS. A common service usage scenario researchers used over the last decade to study service compositions was the *Travel Reservation* application[2]. In this scenario, a travel agent company offers the ability to book complete vacation packages including transportation, accommodation,

[2] http://www.w3.org/2002/04/17-ws-usecase.html

activities, etc. The travel agent company can then select the services it considers the most appropriate to create the composite service that answers its client requests. For example, the travel agent company selects services providing flight ticket reservation, car renting, and hotel booking. Note that these tasks are interrelated, so renting a car and booking a flight may be useless if we cannot find a hotel.

To summarize, we say that a service is an operation exposed in the Web with functional and non-functional properties; in the same way, a composite service is a combination of more than one services which is also exposed in the Web with functional and non-functional properties. In the next section, we present the basic notions about the execution control and fault tolerance mechanisms for composite services.

2.2 Composite Service Execution Control

The execution of a composite service implies the invocation of all component services according to the execution flow imposed by the structure representing the composite service. There exist two basic variants of execution scenarios: sequential and parallel. In a sequential scenario, some services cannot be invoked until the previous ones have finished, because they need the attributes produces by them, or there are restriction controls sequentially imposed. In a parallel scenario, several services can be invoked simultaneously because they do not have data or control flow dependencies.

The execution control of composite services can be centralized or distributed. Centralized approaches consider a coordinator managing the whole execution process [80, 100]. In distributed approaches, the execution process proceeds with the collaboration of several participants without a central coordinator [20, 24]. On the other hand, the execution control could be attached to services [51, 55] or independent of its implementation [25]. Some execution engines are capable of managing failures during the execution. Ones are based on exception handling [65, 83], others are based on transactional properties [24, 34], others use a combination of both approaches [51, 55], while some works base fault tolerance on replication techniques [20, 104]. In this thesis, we use transactional properties as the building block for fault tolerance. The execution control of our system is distributed and implemented using agents which are independent of implementations of services. On top, we implement redundancy and design diversity-based

techniques such as service replacement and replication.

In the next sections, we present the fault hypothesis considered in this thesis, the transactional properties for composite services used as the foundation of our fault tolerance approach, and the main recovery mechanisms that we implement.

2.2.1 Fault Hypothesis

During the execution of a composite service, faults may occur at hardware, operating system, component services, execution engine, and network levels [29]. These faults result in reduced performance and may cause unexpected behavior during a composite service execution. Service faults may be classified as follows:

- Silent or fail-stop faults: these faults are generic to all services and cause services to not respond, because they are not available, or a crash occurred in the platform. Some examples of silent faults are communication timeout, service unavailable, bad gateway, and server error.
- Logic faults: these faults are specific to services, and are caused by error in input attributes (e.g., bad format, out of the valid range, calculation faults) and Byzantine faults (the service still responds to invocation, but wrongly). Moreover, various exceptions thrown by services to the users are classified into the logic-related faults.

The standard ISO 10303-226 [46] defines *fault* as an abnormal condition or defect in a component, equipment, or sub-system level which may lead to a *failure*. Following this definition, in this thesis we consider that:

- component services may fail due to any type of fault;
- we do not distinguish among the types of faults. In our approach, a service either fails or executes successfully;
- the failure probability of a service is related to its availability QoS criterion (Section 2.1);
- we invoke services from a system that runs far from service hosts in reliable servers which do not fail, which data network is highly secure, and which is not affected by service faults, since its execution control is detached from services.

2.2.2 Transactional Properties for Services

In the research field of fault tolerance for composite services, one option to provide fault tolerance is through low level programming constructs such as exception handling [20, 78, 84]. A standard executable language for specifying actions and exception handling for composite services based on SOAP services is WS-BPEL [65], commonly known as BPEL. Exception handling normally is explicitly specified at design time, regarding how exceptions are handled and specifying the behavior of a composite service when an exception is thrown. This approach is normally used to manage logic faults, which are specific to each service, and therefore, specific to the considered composite service.

The reliability of composite services has also been handled at a higher level of abstraction; i.e., at the execution flow structure level, such as workflows or graphs; therefore, technology independent methods for the composition of reliable composite services and their fault tolerant execution have emerged, such as transactional properties [24, 25, 27, 34, 51, 95, 96]. Transactional properties implicitly describe service behavior in case of failures and are used to ensure the traditional atomicity, consistency, isolation, and durability (ACID) properties. When transactional properties are not considered, the ACID properties are the responsibility of users or developers.

The Two-phase Commit (2PC) [35] is a standard protocol in distributed transactions that has been used for achieving ACID properties. It is a distributed consensus algorithm that coordinates all processes participating in a distributed atomic transaction on whether to commit or abort the transaction. However, the 2PC protocol implements resource locking/blocking [35], which is not suitable for long running transactional composite services [27]. As pointed out by Casado et al. [27], researchers have developed advanced models to relax the ACID properties allowing the compensation of completed transitions. Some of these models are: nested transaction [62], SAGA [36], open-nested [94], split-join [72], flex [99], and WebTram [97]. Casado et al. highlighted the following transactional models for services: OASIS Business Transaction Protocol [5], Web Services Business Activity [7], and the Web Service Transaction Management [6]. They also talked about the TQoS model [34], and classified it as “other models and frameworks”. Indeed, TQoS is not SOAP-based and provides QoS-awareness for transactional composite services.

The work presented in this thesis is based on this latter model, TQoS [34], which builds on top of the following transactional properties for elementary services:

- Pivot(p): A service is pivot if its effects remain forever and cannot be semantically undone once it has completed successfully. It is the most basic transactional property.
- Compensable (c): a service is compensable if there exists another service that may semantically undo its successful execution.
- Retriable (r): a service is retriable if it guarantees a successful execution after a finite number of invocations. This property has to be combined with the pivot or compensable properties, creating pivot-retriable (pr) and compensable-retriable (cr) services.

Composite services built from services providing transactional properties guarantee the system consistency and have an aggregated transactional property as follows:

- Atomic (a): a composite service is atomic if at least one of its component services is pivot or pivot-retriable. Once an atomic composite service finishes successfully, its effects remain forever and they cannot be undone. If one of its component services fails, the system is left in a state semantically similar to the one before the execution of the composite service.
- Compensable (c): a composite service is compensable if all its component services are compensable. This means that it exists another composite service, containing the services which compensate the component services of the compensable composite service, which may semantically undo the effects of the compensable composite service after its successful execution. As for the atomic composite service, if one of its component services fails, the system is left in a state semantically similar to the one before the execution of the compensable composite service.
- Retriable (r): a composite service is retriable if all its component services are retriable. A retriable composite service guarantees a successful execution after a finite amount of time. This property has to be combined with the atomic or compensable properties, creating atomic-retriable (ar) and compensable-retriable (cr) composite services.

A composite service must satisfy the graph structure imposed by the transactional model to provide transactional support. Figure 2.3 illustrates some examples of valid and invalid combinations of services with transactional properties.

Directed arcs reflect the execution order between the services. Some examples of invalid transactional properties combinations are the following:

- In Figure 2.3 (a) if the pivot service is executed successfully but the compensable service fails, the system remains in an inconsistent state since the execution of the pivot service cannot be undone.
- Figure 2.3 (b) shows an incorrect combination of a pivot and a pivot-retriable service since they should not be executed in parallel: the pivot-retriable service guarantees a successful execution that cannot be undone; therefore, if the pivot service fails and the pivot-retriable service was successfully executed, the system remains in an inconsistent state.
- Figure 2.3 (c) and Figure 2.3 (d) show examples following the same principle where the system remains in inconsistent states if one of the non-retriable services fails.

Some examples of valid transactional properties combinations are the following:

- In Figure 2.3 (e), if the pivot service is successfully executed, the compensable-retriable service guarantees a successful execution.
- The example of Figure 2.3 (f) is similar to the one of Figure 2.3 (e) since if the compensable service is successfully executed, the pivot-retriable services guarantee successful executions.
- Both Figure 2.3 (c) and Figure 2.3 (d) show valid combination examples where all services are retriable. All retriable services guarantee a successful execution.

Figure 2.4 shows the automaton modeling all possible transactional composite services [34]. It contains five states: I representing the initial state; and c, cr, a, and ar representing the four final states. The final states correspond to the possible transactional properties of a composite service. Transitions between states indicate the transactional property of a service that may be used to perform compositions in sequence (;) or in parallel (//). For example, suppose that we start from the initial state with a pivot service. We reach the final state a, but we will never be able to reach another final state. The reason is that, if we choose a pivot service, not matter with which service we compose it, we will

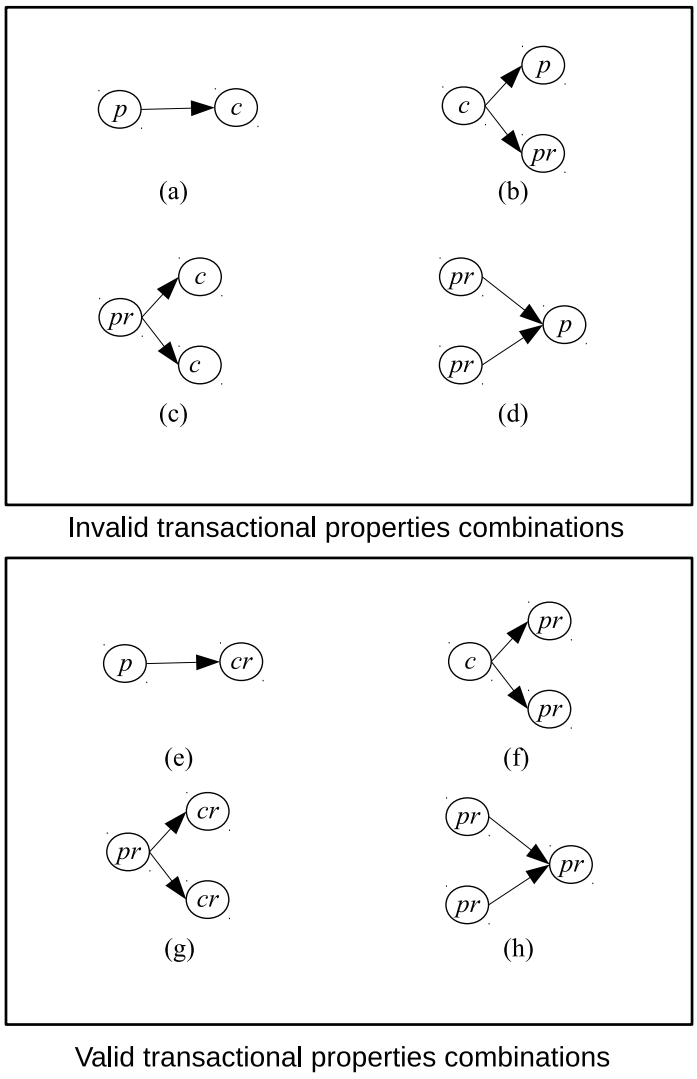


Figure 2.3: Example of Transactional Properties Combinations.

never have a retriable or compensable composite service. However, to ensure the transactional property, a pivot service may only be composed in parallel with a *cr* service, and sequentially with retriable services (*cr*, *pr*, *ar*) as the automaton shows. Composite services built from transactional services must comply with the rules imposed by this automaton to guarantee the system consistency. If a composite service is built from transactional services but it does not comply with the rules of the automaton of Figure 2.4, then it is not a transactional composite service.

Finally, in this thesis we use the presented transactional model to define a compensation protocol to ensure a *relaxed atomicity property* for composite services.

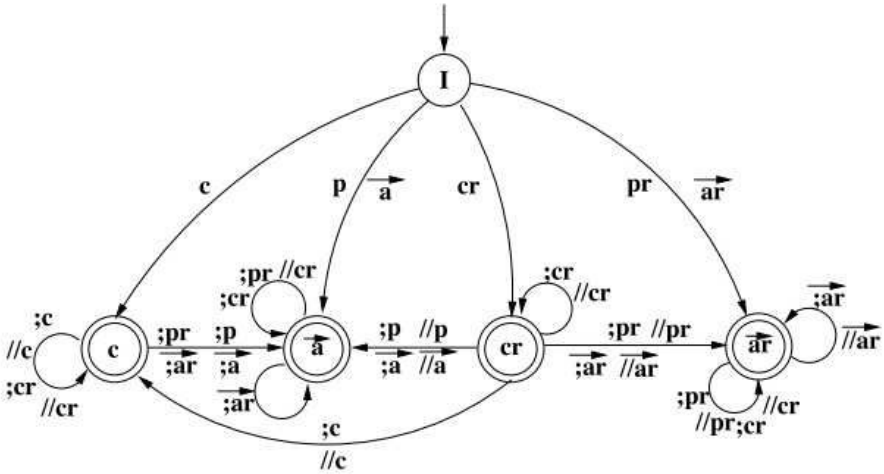


Figure 2.4: Automaton modeling all possible transactional composite services from [34].

This relaxed atomicity property allows the compensation of successfully executed services to leave the system in a state close to the one it had before the execution of those services. It is worth noticing that, according to TQoS, a valid transactional composite service cannot have more than one pivot service. Without this hypothesis, all participating pivot services will need to implement protocols such as 2PC to guarantee atomicity.

2.2.3 Recovery Mechanisms

Services that provide transactional properties are useful to guarantee reliable composite service execution, ensuring the whole system consistent state, even in the presence of failures. Failures during the execution of a composite service may be handled according to the transactional property of its component services by forward or backward recovery mechanisms [25], or by delaying the composite service execution [77]. The main recovery mechanisms are the following:

- Backward recovery: it consists in restoring the state that the system had at the beginning of the composite service execution; i.e., all the effects produced by the failed service are undone through rollback, and the effects of previously executed services are semantically undone by compensation techniques (Fig. 2.5 (a));

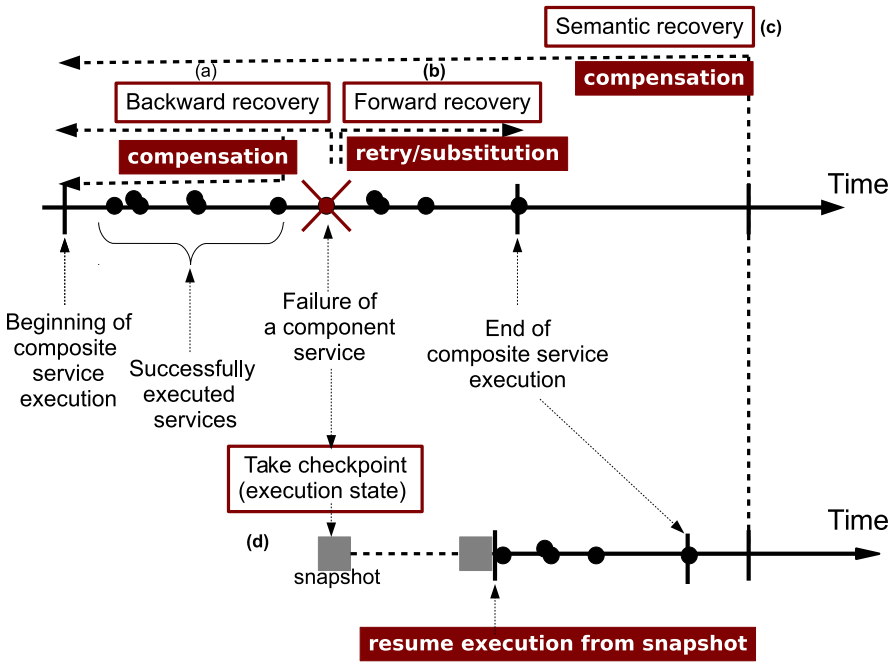


Figure 2.5: Recovery Mechanisms.

- Forward recovery: it consists in repairing the failure to allow the failed service to continue its execution; retry and substitution are some techniques used to provide forward recovery (Fig. 2.5 (b)).
- Semantic recovery: it is similar to backward recovery except that semantic recovery is done after a successful execution of a composite service by compensating the execution of its component services. The idea is to leave the system in a state semantically close to the state it had before the execution of the composite service (Fig. 2.5 (c)).
- Checkpointing: if a failure occurs, it consists of continuing the execution of the part of the composite service not affected by the failure, while delaying the execution of the affected part (Fig. 2.5 (d)).

Due to the great proliferation of services published on the Internet, equivalent services designed/developed independently by different organizations, may be readily employed as redundant alternative components for building diversity-based fault tolerant systems. Equivalent services may be used to allow forward recovery by replacing a failed service regardless of transactional properties.

2.3 Self-healing systems

So far, we have presented the execution control and fault tolerance mechanisms for composite services; however, the goal of this thesis is to provide a smarter execution approach for composite services capable of making decisions depending on what is happening at runtime. For this reason, part of this thesis enhances composite services with self-healing capabilities. We present a brief introduction to self-healing systems in this section.

As we saw in Chapter 1, IBM introduced the notion of self-healing as part of the Autonomic Computing initiative [44, 45, 50]. They stated that the only solution to the complexity crisis [44, 45] was to create computing systems capable of managing themselves. The four main aspects of autonomic computing are self-configuration, self-optimization, self-healing, and self-protection. IBM defined these aspects as follows:

- Self-configuration: it refers to the automatic installation, configuration, and integration of systems.
- Self-optimization: it refers to the automatic improvement of performance and efficiency of systems; for example, by tuning system parameters.
- Self-healing: it refers to the automatic problem and failure detection and recovery.
- Self-protection: it refers to the automatic detection and recovery from attacks.

Later, in a survey of self-healing systems research, Ghosh and his coauthors [37] gave the following definition:

> "Self-healing can be defined as the property that enables a system to perceive that it is not operating correctly and, without (or with) human intervention, make the necessary adjustments to restore itself to normalcy."

Furthermore, Ghosh and his coauthors [37] introduced the self-healing model illustrated in Figure 2.6 to highlight the importance of understanding normal vs.

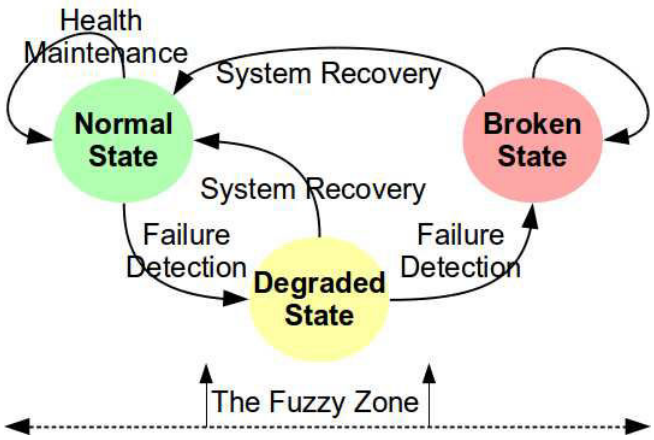


Figure 2.6: Self-healing States (taken from [37]).

abnormal behavior when designing and studying complex systems. Figure 2.6 depicts the three possible states of a self-healing system and the transitions between them. The "degraded" state refers to a fuzzy zone where there is no clear distinction between a "normal" and an "broken" state. This fuzzy zones models the fact that large scale distributed systems are composed by modular components; therefore; if a small part of the system fails, the rest should be able to continue its operations without disruption, while the fault tolerance mechanisms try to fix the detected problems and go back to the "normal" state.

The health maintenance transition refers to the constant checking and maintenance of the normal functionality of the system. One common method for maintaining system health is to provide redundancy for the system components. The system can transition from a "normal" state to a "degraded" state by detecting a failure, but it can go back to a "normal" state by repairing the failure. In the same way, the system can transition from a "degraded" state to a "broken" state by detecting a failure, and go to a "normal" state by repairing the failure. The system can also remain in a "broken" state if the failure is not repaired.

Later on in this thesis, we use the model illustrated in Figure 2.6 as a base to describe the self-healing properties for composite services; that is, we define the normal, degraded, and broken states for composite services. Then, we specify which actions should be taken at runtime based on the current self-healing state of the composite service and information about the execution context.

2.4 Petri Nets and Colored Petri Nets

Previously in this chapter, we have presented a composite service as a combination of several services to produce a more complex service. A composite service concerns which and how elementary services are combined to obtain the desired result, and it can be modeled using, for example, the Business Process Execution Language (BPEL) [2] for SOAP Web services, or Petri nets [70]. In this thesis, we extend Colored Petri nets to model the structure and behavior of composite services; hence, we begin by presenting the Petri net formalism; then, we present the colored Petri net formalism. Analysis methods for Petri nets were left out of this introduction.

A Petri net [70] is an abstract, formal model of information flow. It is a mathematical modeling language useful to describe and analyze distributed systems with asynchronous and concurrent activities. A Petri net is a directed, connected, and bipartite graph in which each node is either a *place* or a *transition*. Places represent states or conditions; transitions represent actions. Arcs in a Petri net run from a place to a transition or vice versa, but never between places or between transitions. Formally, we can define a Petri net as follows:

Definition 2.4.1 ***Petri net (PN).*** *A Petri net is a 3-tuple PN=* (D, S, F)*, where:*

- D *is a finite non-empty set of places;*
- S *is a finite set of transitions;*
- $F \supseteq (D \times S) \cup (S \times D)$ *is a set of arcs representing the flow relation between places and transitions.*

To model systems, Petri nets use a notion of *marking* to represent the system state at a given moment, and transition *firing* to the represent the system behavior. A marking in a Petri net is an assignment of *tokens* to the places of the Petri net. Tokens reside in the places of the Petri net. The number and position of tokens may change at runtime; therefore, tokens are used to define and control the execution of a Petri net. More formally, a marked Petri Net is defined as follows:

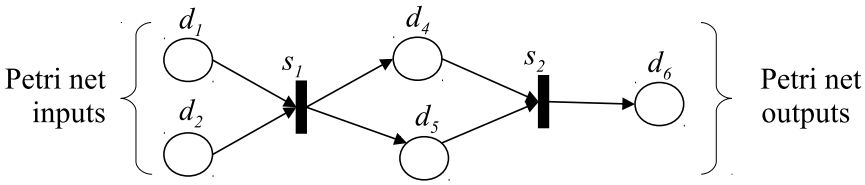


Figure 2.7: Example Petri net.

Definition 2.4.2 ***Marked Petri net.*** *A marked Petri net is a 3-tuple* (PN, W, M)*, where:*

- PN *is a Petri net;*
- $W : F \rightarrow \{1, 2, 3, ...\}$ *is a weight function of arcs.*
- $M : D \rightarrow \{1, 2, 3, ...\}$ *is a function that assigns tokens to places.*

The weight function W for arcs specifies how many tokens an input place must have so that its consumer transition is executed, and how many tokens are set into the output places of an executed transition. The default value of W is 1. A marking of M represents the state of a Petri net at a particular moment; that is, tokens assigned to places.

In our context, places in D represent service inputs and outputs, while transitions in S represent the services. When two services are connected by a place, that place represents an output produced by one of the services and consumed by the other service. Each service sets only one token to its output places; therefore, $w(s, d) = 1$ for all output places d of a transition s. Figure 2.7 shows an example Petri net composed by the service s_1 which inputs are the places d_1 and d_2, and which outputs are the places d_4 and d_5, and the service s_2 which inputs are the places d_4 and d_5, and which output is the place d_6. Places d_1 and d_2 do not have predecessors, they are the places representing the composite service inputs. The place d_6 does not have successors, it represents the output of the composite service.

When a transition has the required tokens (defined by the W function) in its inputs places to be executed, we say that the transition is *fireable*, as follows:

Definition 2.4.3 ***Fireable Transition.*** *A transition* s *is said to be fireable if each input place* d *of* s *is marked with at least* $w(d, s)$ *tokens, where* $w(d, s)$ *is the*

weight of the arc from d to s; that is, a transition is enable if for all its input places:

$$M(d) \geq w(d, s)$$

In our context, we define the *initial marking* M_0 of a Petri net as the state when all input places have tokens and the rest of the places do not have tokens. Returning to our example, Figure 2.8 (a) shows the initial marking M_0 of a Petri net when both input places, d_1 and d_2, have tokens, while the rest of the places do not have tokens. In other words, the transition s_1 is fireable in the Petri net state showed in Figure 2.8 (a) since Def. 2.4.3 satisfies as follows:

$$M(d_1) \geq 1 \ \wedge \ M(d_2) \geq 1$$

When a transition is fired, it consumes the required token from its input places, it executes, and it puts tokens in its output places according to the weight function W, as follows:

Definition 2.4.4 ***Firing of a Transition.*** *The firing of a transition s removes $w(d, s)$ tokens from each input place d of s, and adds $w(s, d)$ tokens to each output place d of s, where $w(s, d)$ is the weight of the arc from s to d; that is, for all input places d of transition s:*

$$M'(d) = M(d) - w(d, s)$$

and for all output places d of transition s:

$$M'(d) = M(d) + w(s, d)$$

Figure 2.8 (b) shows the marking M' of the Petri net after s_1 has been fired and the output places d_4 and d_5 contain their corresponding tokens. By Def. 2.4.4, we have that:

$$M'(d_1) = M(d_1) - w(d_1, s_1) = 1 - 1 = 0$$
$$M'(d_2) = M(d_2) - w(d_2, s_1) = 1 - 1 = 0$$
$$M'(d_4) = M(d_4) + w(s_1, d_4) = 0 + 1 = 1$$
$$M'(d_5) = M(d_5) + w(s_1, d_5) = 0 + 1 = 1$$

Then, s_2 becomes fireable since:

$$M(d_4) \geq 1 \ \wedge \ M(d_5) \geq 1$$

Similarly, Figure 2.8 (c) shows the marking M' of the Petri net after s_2 has been fired and the output place d_6 contains its corresponding token.

An illustration of modeling composite services as Petri nets is the work presented by Hamadi and Benatallah in 2003 [41]. In their work, they propose a Petri net-based algebra to model control flows as a necessary constituent of reliable service compositions. As part of the conclusions of their work, they state that some aspects of composite services such as the management of time and resources cannot be modeled with Petri nets, but have to be dealt by using a suitable high-level Petri net, such as Colored Petri nets. Usually, in systems modeled by Petri nets, tokens represent objects or resources that may have attributes that cannot be represented by a simple Petri net token.

The Colored Petri net [48] formalism was introduced by Jensen in the 80's to extend the Petri net model. Using this formalism, information can be modeled by tokens and the type of information can be modeled by the color of those tokens. This Petri net extension allows the attachment of data values to tokens in contrast to Petri nets where we have only plain tokens with no additional information. Data value attached to tokens belong to a given *type*.

In this thesis, we use colors to model the type of input and output places of transitions. We associate these Colored Petri net types with concepts of an ontology, as we showed in Section 2.1 for service input and outputs. Additionally, we incorporate the notion of colors for transitions to represent transactional properties, and additional information to control the execution of Petri nets.

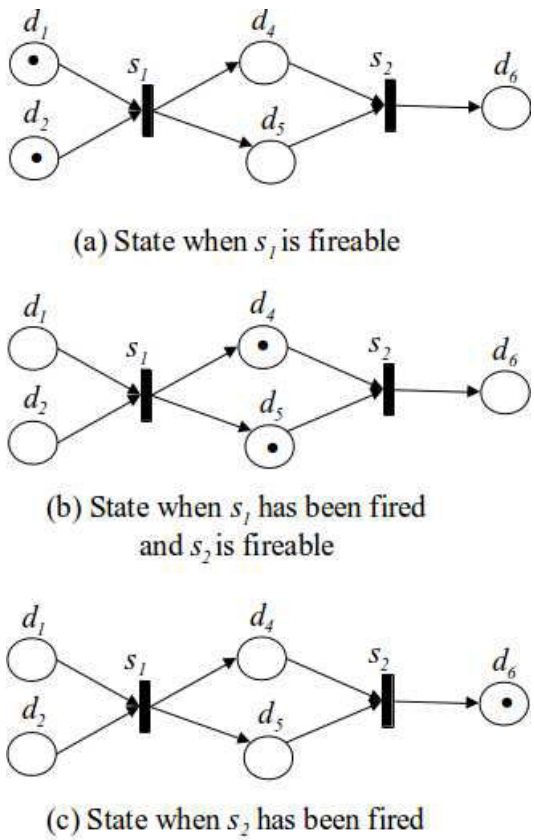


Figure 2.8: Fireable Transition Example.

2.5 Summary of General Assumptions

In this section, we summarize some general assumptions presented in this chapter. We consider these assumptions address issues beyond the scope of this thesis since they concern different, although related, research areas. The following are the general assumptions considered in this thesis:

1) **System consistency**: we assume we are dealing with applications that require transactional support. Therefore, services need to be combined to create transactional composite services to guarantee the system consistency.

2) **Service discovery and selection**: we suppose services are published in a machine or human oriented registry, they have already been discovered, and classified by functionality. Therefore, services are ready to be selected and used by our system.

3) **Service composition**: composite services may be generated by either an automatic composition process such as [34] or manually; however, their structure always satisfy the definition of transactional composite services given in Section 2.2.2.

4) **Transactional properties**: all our services are transactional, so they are annotated with their corresponding transactional properties.

5) **Service implementation**: services may be implemented following the SOAP standard, the RESTful architectural style, or any other technology. We suppose we are able to dynamically invoke any service independently of the underlying technologies.

In the next chapter, we build on top of the Colored Petri net formalism to present an approach for composite service fault tolerance.

Chapter 3

Composite Service Execution Control and Recovery Mechanisms

Contents

3.1 Modeling composite service executions 34

3.1.1 Backward Recovery 42

3.1.2 Checkpointing 48

3.1.3 Service Replacement 52

3.2 Framework Architecture 57

3.2.1 Fault Tolerance Algorithms 60

3.3 Conclusions 69

This chapter presents our fault tolerance approach for composite service execution. Our approach allows the efficient, fault tolerant, and correct execution of composite services. It implements two main fault tolerance mechanisms: forward and backward recovery. Forward recovery is done through service replacement and service retry. Backward recovery is done by implementing a compensation protocol. Additionally, we present a checkpointing mechanism to continue the execution of the part of a composite service not affected by a failure, while delaying the execution of the affected part.

The content presented in this chapter is the continuation of the work of Cardinale and Rukoz [25], where they presented an approach for the modeling of fault

	Description
CPN-EP	A Colored Petri net representing the execution plan of a composite service.
BRCPN-EP	A backward recovery Colored Petri net representing a compensation execution plan.
CPN-EP_{IN}	Colored Petri net inputs.
CPN-EP_{OUT}	Colored Petri net outputs.
CPN-$EP_{snapshot}$	A checkpointed Colored Petri net.
CPN-EP'_{IN}	Inputs of a checkpointed Colored Petri net.
CPN-EP'_{OUT}	Outputs of a checkpointed Colored Petri net.
s	A transition representing a service.
$s_{\equiv}$	Set of equivalent services of s.
s^*	A replacement service for s.
s'	A compensation service for s.
s'^*	A replacement service for a compensation servcice s'.
S	A set of services.
D	Set of places in a Colored Petri net representing service inputs and outputs.
d	A place in a Colored Petri net representing service inputs and outputs.
γ	A Service Agent.
$s_{\Diamond}$	A control node representing the beginning of a composite service.
$s_{\blacklozenge}$	A control node representing the end of a composite service.
Γ	The set of all Service Agents.
${}^{\bullet}s$, ${}^{\bullet}\gamma$	Inputs of service s/Service Agent γ.
${}^{\bullet}({}^{\bullet}s)$, ${}^{\bullet}({}^{\bullet}\gamma)$	Predecessor services of service s/Service Agent γ.
$s^{\bullet}$, $\gamma^{\bullet}$	Outputs of service s/Service Agent γ.
$(s^{\bullet})^{\bullet}$, $(\gamma^{\bullet})^{\bullet}$	Successor services of service s/Service Agent γ.
p, a, pr, ar, c, cr	The available transactional properties for services.
$TP(s)$	Transactional property of a service s with $TP(s) \in \{p, a, pr, ar, c, cr\}$.
$QoS(s)$	QoS of a service s.
$d \subseteq_{is-A} d'$	Data d represents a concept semantically equal or a subconcept of d'.
d_{value}	The actual value of a place of type d.
$\bullet$	Data value for places in a CPN-EP.
$\circ$	Control tokens for places in a CPN-EP and BRCPN-EP.
x	Skip control tokens for places in a CPN-EP.

Table 3.1: Chapter 3 Notation.

tolerant composite services with Colored Petri nets. In particular, in this chapter we refine the initial definitions proposed in [25], extend them to model the checkpointing mechanism, and improve the service replacement approach. The notation used in this chapter is presented in Table 3.1.

3.1 Modeling composite service executions

In this thesis, we model composite services and their execution process using the Colored Petri net [48] formalism. In Section 2.4, we showed that a Colored

Petri net is an abstract, formal model of information flow useful to describe and analyze distributed systems with asynchronous and concurrent activities. We suppose that our composite services modeled using Colored Petri nets do not contain cycles; that is, there is no transition reachable from itself.

We introduce the set of definitions of this chapter with the formal definition of a composite service using the Colored Petri net formalism:

Definition 3.1.1 ***Composite Service (CPN-EP).*** *A composite service CPN-EP is a 4-tuple (D, S, F, ξ), where:*

- *D is a finite non-empty set of places, corresponding to input and output attributes of the component services;*
- *S is a finite set of transitions corresponding to the set of services in the CPN-EP;*
- *$F : (D \times S) \cup (S \times D) \rightarrow \{0, 1\}$ is a function establishing the flow relation between places and transitions defined as: $\forall s \in S$, $\exists d \in D \mid F(d, s) = 1 \Leftrightarrow$ d is an input place of s; and $\forall s \in S$, $\exists d \in D \mid F(s, d) = 1 \Leftrightarrow d$ is an output place of s;*
- *ξ is a color function such that ξ: $C_D \cup C_S$, with:*
  - *C_D: $D \rightarrow \sum_D$, a color function such that $\sum_D = \{DATA, CTRL\}$ representing the two types of tokens for places in the CPN-EP: $DATA$ represents the data types, associated to an ontology, of inputs and output attributes of services, and it is visually represented by a black circle "●"; $CTRL$ represents constant values to control the execution of the CPN-EP. The possible value of $CTRL$ is:*
    - *$CTRL_TOKEN$: controls the normal execution flow of a CPN-EP, and it is visually represented by an empty circle "○".*
  - *$C_S : S \rightarrow \sum_S$, a color function such that $\sum_S = \{p, pr, a, ar, c, cr\}$ represents the transactional property of the corresponding service of transition s (see Section 2.2.2).*

From now on, we use the notation s to refer indistinctly to transitions and services. Def. 3.1.1 presents a composite service model using the Colored Petri net formalism; however, note that workflows, bipartite graphs, etc, can be matched to

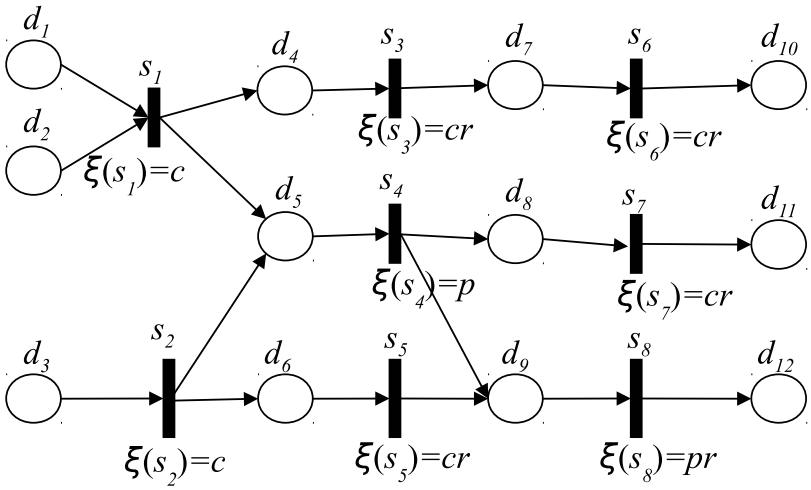


Figure 3.1: A Composite Service CPN-EP.

our composite service definition, even if the relationship among services is defined by data flow or control flow. In Figure 3.1, we show a composite service modeled by a Colored Petri net. This Colored Petri net is composed of 8 services and 12 input and output attributes, as follows:

$$D = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}\}$$
$$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$$

We can see that the transactional property of service s_4 is pivot, so the transactional property of the whole composition is atomic, as explained in Section 2.2.2. Predecessor services of s_4 are compensable, its successors are retriable, and its parallel services are compensable-retriable, satisfying the transactional model for composite services.

Note that the function F of Def. 3.1.1 may represent both data flow and control flow. Data flow represents functional dependency between two services; that is, an input needed by a service is the output produced by another service. For example, in Figure 3.1, place d_4 is part of the data flow dependency between s_1 and s_3. The purpose of control flow is to make sure the CPN-EP satisfies the transactional model presented in Section 2.2.2. An example of both data and control flow dependency is the input place of s_8, d_9. Since the transactional property of s_8 is pivot-retriable, all non-compensable and non-retriable services must be executed before it. That is why there is a control flow dependency

between s_4 and s_8. Without it, s_8 may be successfully executed before a failure of s_4 making backward recovery through compensation impossible, and leaving the system in an inconsistent state. Moreover, this control flow restriction works both if s_4 produces a control token or a data value for d_9.

Regarding the accepted values by places, Def. 3.1.1 states that places may contain any value which type is a subtype of the data type assigned to the place following the relation $\subseteq_{is-A}$ (Section 2.1), or control tokens $CRTL$.

Places with no predecessors represent the inputs of the composite service, while places with no successors represent its outputs. The following is the formal definition of the inputs and outputs of a composite service represented by a CPN-EP:

Definition 3.1.2 ***Composite Service Inputs (CPN-EP_{IN}) and Outputs (CPN-EP_{OUT}).*** *The inputs of a composite service CPN-$EP(D, S, F, \xi)$ are a subset CPN-EP_{IN} of D such that:*

$$\forall d \in D,\ d \in \textit{CPN-}EP_{IN} \Leftrightarrow\ {}^{\bullet}d = \emptyset$$

The outputs of a composite service CPN-$EP(D, S, F, \xi)$ are a subset CPN-EP_{OUT} of D such that:

$$\forall d \in D,\ d \in \textit{CPN-}EP_{OUT} \Leftrightarrow\ d^{\bullet} = \emptyset$$

Back to our example, the inputs and outputs of the composite service are:

$$\text{CPN-}EP_{IN} = \{d_1, d_2, d_3\}$$
$$\text{CPN-}EP_{OUT} = \{d_{10}, d_{11}, d_{12}\}$$

To control the execution of a composite service, we define initial $s_{\Diamond}$ and final $s_{\blacklozenge}$ transitions, which are added to the CPN-EP. These transitions have only control responsibilities to define the start and the end of composite services. We define them as follows:

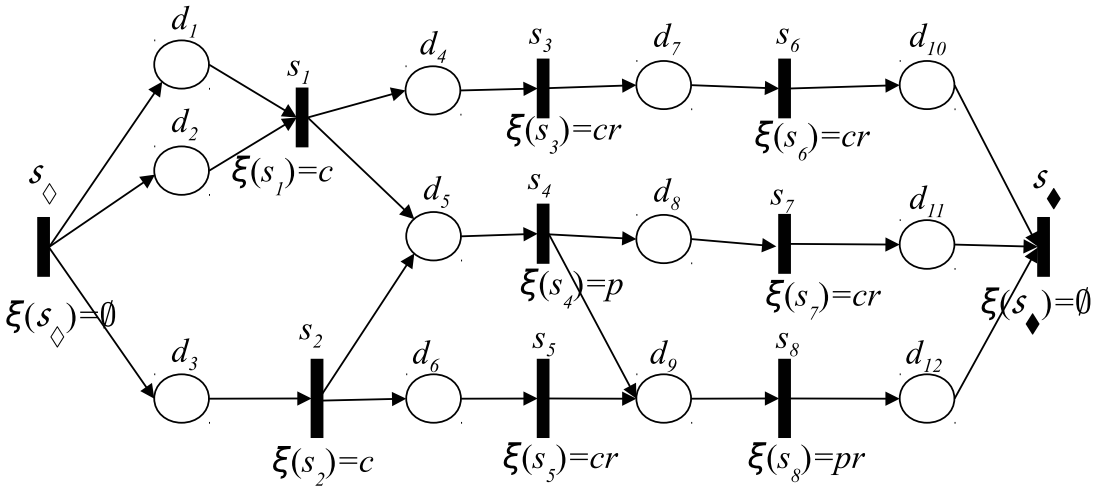


Figure 3.2: A Composite Service with Initial and Final Transitions.

Definition 3.1.3 ***Initial and final transitions of a composite service.*** *Let CPN-EP be a composite service; the initial and final transitions, denoted as* $s_\diamondsuit$ *and* $s_\blacklozenge$ *respectively, are dummy transitions added to a composite service, such that:*

- $S \leftarrow \{s_\diamondsuit, s_\blacklozenge\} \cup S$;
- $\forall\ d \in D,\ {}^\bullet d = \emptyset\ \rightarrow F(s_\diamondsuit, d) = 1$. $s_\diamondsuit$ *is the predecessor transition of all input places of CPN-EP;*
- ${}^\bullet s_\diamondsuit \leftarrow \emptyset$;
- $\forall\ d \in D,\ d^\bullet = \emptyset\ \rightarrow F(d, s_\blacklozenge) = 1$. $s_\blacklozenge$ *is the successor transition of all output places of CPN-EP;*
- $s_\blacklozenge^\bullet \leftarrow \emptyset$;
- $\xi(s_\diamondsuit) \leftarrow \emptyset$;
- $\xi(s_\blacklozenge) \leftarrow \emptyset$.

As showed in Def. 3.1.3, $s_\diamondsuit$ is unconditionally fireable, while $s_\blacklozenge$ consumes tokens but do not produce any. Figure 3.2 shows our composite service example with its corresponding initial and final transitions. This way, the starting of the composite service execution is controlled by $s_\diamondsuit$, and the firing of $s_\blacklozenge$ means the composite service execution has finished.

A marking of a CPN-EP represents the execution state of the CPN-EP at a particular moment. A marking contains the current values of attributes that may be used for some component services to be executed, or control values indicating the execution flow.

Definition 3.1.4 ***Marked executable CPN-EP.*** *A marked CPN-EP=(D, S, F, ξ) is a pair (CPN-EP,M), where M is a function which assigns tokens (values) to places such that $\forall d \in D,\ M(d) \subseteq \{\emptyset, Bag(\sum_D)\}$, where Bag corresponds to a multiset which may contain several occurrences of the same element. The marking of a CPN represents the current state of the system, i.e., the set of attributes correctly produced by the system and/or signals indicating failures.*

A transition s is fireable when all its input places contain a token per predecessor transition. If all the input places of s contain data values ($DATA$) and/or control tokens with value $CTRL_TOKEN$, we say that s is fireable. The following is the definition of a fireable transition:

Definition 3.1.5 ***Fireable transition.*** *A marking M enables a transition s for execution if and only if all its input places contain tokens such that*

$$\forall d \in {}^{\bullet}s,\ card(M(d)) = card({}^{\bullet}d)\ \wedge \\ M(d) \subseteq (Bag(\{DATA\} \cup Bag(\{CTRL_TOKEN\}))$$

The execution control of a composite service is guided by an unrolling algorithm of its corresponding CPN-EP. To start the unrolling algorithm, transition $s_\Diamond$ is fired.

The firing of a transition of a CPN-EP corresponds to the execution of the service or composite service represented by that transition. Then, when s finishes, it is considered that the transition was fired, and other transitions may become fireable. In our example, when $s_\Diamond$ is fired, tokens are added to places d_1, d_2, and d_3; therefore, s_1 and s_2 become fireable, and the composite service execution starts. Formally, the firing rules for a CPN-EP are the following:

Definition 3.1.6 ***CPN-EP Firing rules.*** *The firing for execution of a fireable transition s for a marking M defines a new marking M', such that all tokens are deleted from its input places:*

$$\forall d \in {}^{\bullet}s,\ M(d) \leftarrow \emptyset$$

and the service s is invoked. After service s finishes, data value tokens are added to its output places. These tokens contain the actual produced value d_{value} *for a type d:*

$$\forall d \in s^{\bullet}, (M(d) \leftarrow M(d) \cup \{d_{value}\})$$

These actions are atomically executed.

To model the internal behavior of a transition s_i, we add a transition $s_{i-INVOKE}$ representing the invocation of s_i and a place $d_{i-STATE}$ representing the execution state of s_i. $d_{i-STATE}$ is added as output place of $s_{i-INVOKE}$. This way, we extend the CPN-EP such that:

$$\begin{aligned}
&\forall s_i \in S,\ S \leftarrow S \cup \{s_{i-INVOKE}\}\\
&\forall d_i \in {}^{\bullet}s_i,\ F(d_i, s_{i-INVOKE}) = 1\\
&\forall s_i \in S,\ D \leftarrow D \cup \{d_{i-STATE}\}\\
&\forall s_i \in S,\ F(s_{i-INVOKE}, d_{i-STATE}) = 1
\end{aligned}$$

Then, other internal transitions may use the results in $d_{i-STATE}$ to execute actions depending of the execution of s_i. We introduce the transition s_{i-OK} to model a successful execution of s_i, and $s_{i-RETRY}$ to model the reexecution of retriable services. These transitions are added to the CPN-EP as follows:

$$\begin{aligned}
&\forall s_i \in S,\ S \leftarrow S \cup \{s_{i-OK}\}\\
&\forall s_i \in S,\ S \cup \{s_{i-RETRY}\} \text{ if } TP(s_i) \in \{pr, ar, cr\}
\end{aligned}$$

Then, s_{i-OK} and $s_{i-RETRY}$ become successors of $d_{i-STATE}$, and s_{i-OK} becomes the predecessor of all output places of s_i as follows:

$$\forall s_i \in S,\ F(d_{i-STATE}, s_{i-OK}) = 1$$
$$\forall d_i \in s_i^{\bullet},\ F(s_{i-OK}, d_i) = 1$$
$$\forall s_i \in S,\ F(d_{i-STATE}, s_{i-RETRY}) = 1 \text{ if } TP(s_i) \in \{pr, ar, cr\}$$

The transition $s_{i-RETRY}$ does not have successors. The firing of $s_{i-INVOKE}$ means that it removes tokens from its input places, and it sets data values or control tokens to $d_{i-STATE}$. When s_{i-OK} is fired, it removes tokens from $d_{i-STATE}$ and sets data values to its corresponding output places:

$$M(d_{i-STATE}) \leftarrow \emptyset$$
$$\forall d \in {s_{i-OK}}^{\bullet},\ (M(d) \leftarrow M(d) \cup \{d_{value}\})$$

The firing of $s_{i-RETRY}$ means that s_i is retried. This action neither removes nor produces tokens.

Figure 3.3 illustrates these internal mechanisms taking the transition s_3 as example. The firing of s_3 is decomposed as follows:

1) if s_3 executes successfully, s_{3-OK} is fired, $d_{3-STATE} \leftarrow \emptyset$, and data values are added to d_7

2) If the execution of s_3 was not successful, $s_{3-RETRY}$ becomes fireable, and retries s_3 until it executes successfully. Then, s_{3-OK} becomes fireable.

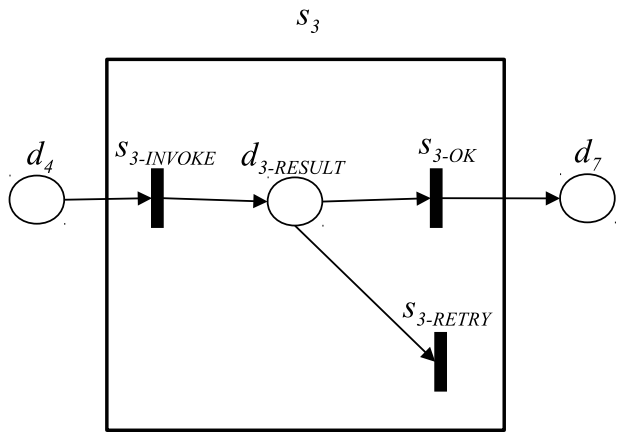


Figure 3.3: Internal Mechanisms of a Transition.

Note that, throughout this thesis, we do not show the internal mechanisms of transitions when illustrating whole composite services. The reason of not showing them is to simplify the representation of our model.

3.1.1 Backward Recovery

The global transactional property of CPN-EP ensures that if a component service whose transactional property does not allow forward recovery fails, then all previous executed services can be semantically recovered by a backward recovery of the composite service. To model the backward and semantic recovery process, we define a Colored Petri net, called BRCPN-EP, associated to a CPN-EP as follows:

Definition 3.1.7 ***BRCPN-EP.*** *A BRCPN-EP, associated to a given CPN-EP = (D,S,F,ξ), is a 4-tuple* (D', S', F^{-1}, ζ)*, where:*

- D' *is a finite set of places, associated to the CPN-EP places such that:* $\forall d' \in D'$, $\exists d \in D$ *associated to* d'*, and* d' *has the same semantic meaning of* d*;*
- S' *is a finite set of transitions representing compensation services associated to compensable services in CPN-EP such that:* $\forall s \in S$, $TP(s) \in \{c, cr\}$, $\exists s' \in S'$ *which compensates* s*;*

- $F^{-1} : (D \times S) \cup (S \times D) \rightarrow \{0, 1\}$ *is a flow relation establishing the restoring order for backward recovery defined as:* $\forall s' \in S'$ *associated to* $s \in S$, $\exists d' \in D'$ *associated to* $d \in D |\ F^{-1}(d', s') = 1 \iff F(s, d) = 1$ *and* $\forall s' \in S'$, $\exists d' \in D' |\ F^{-1}(s', d') = 1 \iff F(d, s) = 1$;
- ζ *is a color function such that* ζ: $C_{D'} \cup C_{S'}$, *with:*
  - $C_{D'}$: $D' \rightarrow \sum_{D'}$, *a color function such that* $\sum_{D'} = \{CTRL\}$ *representing constant values to control the execution of the BRCPN-EP. The only accepted value for* $CTRL$ *is* $CTRL_TOKEN$, *which is visually represented by an empty circle "○";*
  - $C_{S'}$: $S' \rightarrow \sum_{S'}$, *a color function such that* $\sum_{S'} = \{I, R, E, C, A\}$, *representing the execution state of the transition in CPN-EP corresponding to* $s' \in S'$, *with* I: *initial,* R: *running,* E: *executed,* C: *compensated, and* A: *abandoned.*

As we showed in the previous section, the execution control of a composite service is guided by an unrolling algorithm of its corresponding CPN-EP. To support backward recovery, it is necessary to keep the trace of the execution on the BRCPN-EP. To start the unrolling algorithm, the CPN-EP is marked with the Initial Marking of Def 3.1.8, and the state of all transitions in BRCPN-EP is set to "initial":

$$\forall s' \in S',\ \zeta(s') \leftarrow I$$

While a service s in CPN-EP is executing, if $TP(s) \in \{c, cr\}$, the state of its corresponding service s' in BRCPN-EP is set to "running":

$$\zeta(s') \leftarrow R$$

Then, when s finishes, it is considered that the transition was fired following the rules of Def. 3.1.6, and the state of its associated transition s' is set to executed:

$$\zeta(s') \leftarrow E$$

To handle failures using the backward recovery mechanism, the compensation control of a CPN-EP is guided by an unrolling algorithm of its associated

BRCPN-EP. When a service represented by a transition s fails, and the recovery mechanism to apply is compensation, backward recovery is initiated with the unrolling process on the BRCPN-EP. To control the backward recovery process, we add to the BRCPN-EP initial and final transitions, $s'_{\diamond}$ and $s'_{\blacklozenge}$, similar to Def 3.1.3. Additionally, a BRCPN-EP contains initial and final places, $d'_{\diamond}$ and $d'_{\blacklozenge}$, such that:

- $D' \leftarrow \{d'_{\diamond}, d'_{\blacklozenge}\} \cup D'$;
- $F(d'_{\diamond}, s'_{\diamond}) \leftarrow 1$;
- ${}^{\bullet}d'_{\diamond} \leftarrow \emptyset$;
- $F(s'_{\blacklozenge}, d'_{\blacklozenge}) \leftarrow 1$;
- ${}^{\bullet}d'_{\blacklozenge} \leftarrow \emptyset$.

Then, the BRCPN-EP unrolling process may be initiated by placing a control token in $d'_{\diamond}$, as follows:

Definition 3.1.8 ***BRCPN-EP Initial Marking.*** *This initial marking of a BRCPN-EP means that a token is added to the input place of the BRCPN-EP:*

$$M(d'_{\diamond}) \leftarrow \{CTRL_TOKEN\}$$

and other places have no tokens.

We define a fireable compensation transition as follows:

Definition 3.1.9 ***Fireable compensation transition.*** *A marking M enables a transition s' for compensation if and only if all its input places contain tokens such that*

$$\forall d' \in^{\bullet} s',\ M(d') \neq \emptyset \wedge \zeta(s') \notin \{A, C\}$$

where A means abandoned, and C means compensated.

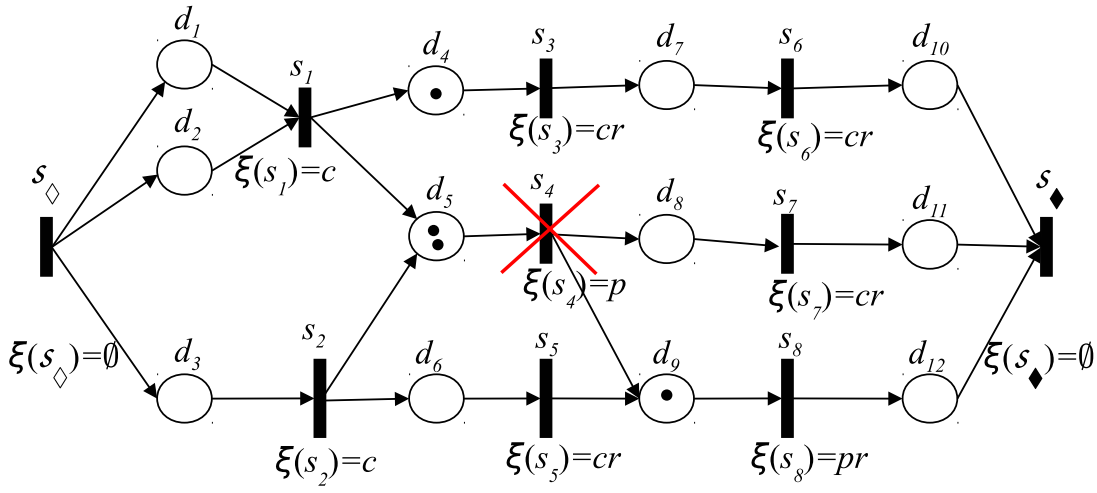


Figure 3.4: CPN-EP example when s_4 fails.

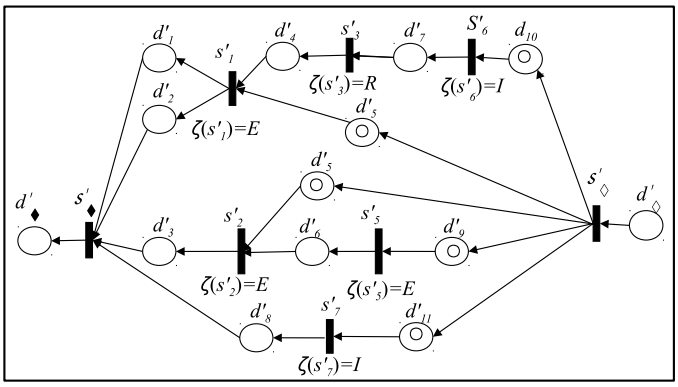
The following are the firing rules for a transition s' in BRCPN-EP:

Definition 3.1.10 ***BRCPN-**EP* ***Firing rules.*** *The firing of a fireable compensation transition (see Def. 3.1.9) s' for a marking M defines a new marking M', such that:*

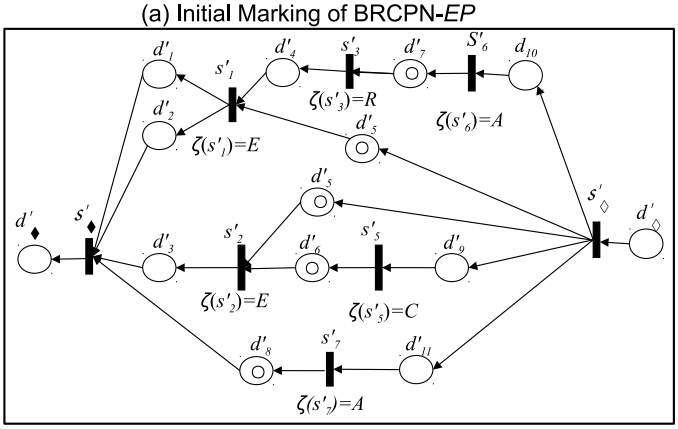
- *if $\zeta(s') = I$, $\zeta(s') \leftarrow A$ (i.e., the associated transition s was abandoned before its execution);*
- *if $\zeta(s') = R$, $\zeta(s') \leftarrow C$ (in this case, s' will be executed after s finishes);*
- *if $\zeta(s') = E$, $\zeta(s') \leftarrow C$ (in this case, s' is executed);*
- *all tokens are deleted from its input places ($\forall d \in {}^{\bullet}s'$, $M(d) = \emptyset$) and tokens are added to its output places ($\forall d \in s'^{\bullet}$, $M(d) \leftarrow M(d) \cup \{CTRL_TOKEN\}$)*

Then, the fireable compensation transition defined in Def. 3.1.9, and the firing rules defined in Def. 3.1.10, guide the unrolling process of the BRCPN-EP.

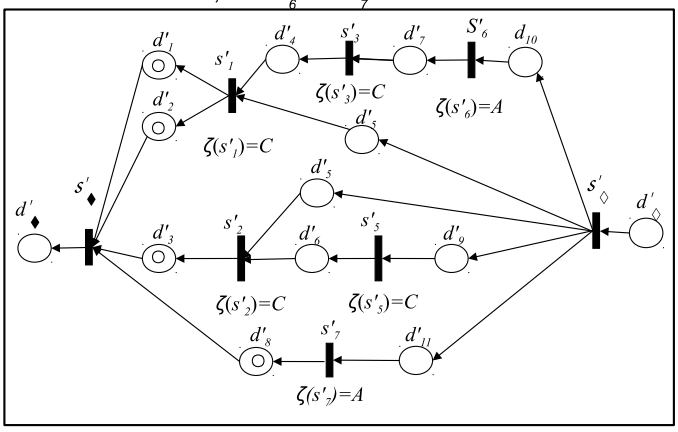
To illustrate the backward recovery process, let us consider the marked CPN-EP represented in Figure 3.4 which is the state of the CPN-EP when s_4 fails. When the failure of s_4 is detected, the backward recovery process is initiated. The corresponding initial marking on the BRCPN-EP is established to start its unrolling process (Figure 3.5 (a)). Then, s'_6 and s_7 are abandoned, s'_5 is invoked to compensate s_5, and a new marking is produced (Figure 3.5 (b)) in which s'_3 and



(a) Initial Marking of BRCPN-*EP*



(b) BRCPN-*EP* Marked after s'_5 was invoked, and s'_6 and s_7 were abandoned



(c) End of BRCPN-*EP* unrolling

Figure 3.5: Backward Recovery Example.

s'_2 are both fireable and can be invoked in parallel. Finally, Figure 3.5 (c) shows the BRCPN-*EP* sate when the backward recovery process has finished. Note that only compensable transitions have corresponding compensation transitions in the BRCPN-*EP*.

Regarding the internal mechanism of compensation, we add the transition $s_{i-COMPENSATE}$ for services which are not retriable as follows:

$$\forall s_i \in S,\ S \leftarrow S \cup \{s_{i-COMPENSATE}\} \text{ if } TP(s_i) \notin \{pr, ar, cr\}$$
$$\forall s_i \in S,\ F(d_{i-STATE}, s_{i-COMPENSATE}) = 1 \text{ if } TP(s_i) \notin \{pr, ar, cr\}$$
$$\forall s_i \in S,\ F(s_{i-COMPENSATE}, d'_{\Diamond}) = 1 \text{ if } TP(s_i) \notin \{pr, ar, cr\}$$

This way, a non-retriable service is internally composed by the transitions $s_{i-INVOKE}$, s_{i-OK}, and $s_{i-COMPENSATE}$; a retriable service is internally composed by $s_{i-INVOKE}$, s_{i-OK}, and $s_{i-RETRY}$.

When a service s_i fails and $TP(s_i) \notin \{pr, ar, cr\}$, $s_{i-COMPENSATE}$ is fired, it removes tokens from s_{i-OK} and sets a control token to its output place, $d'_{\Diamond}$, to initiate backward recovery as follows:

$$M(s_{i-OK}) \leftarrow \emptyset$$
$$M(d'_{\Diamond}) \leftarrow \{CTRL_TOKEN\}$$

Figure 3.6 illustrates the internal mechanisms of a non-retriable transition using s_4 as example. If s_4 fails, $s_{4-COMPENSATE}$ removes the token from $d_{4-STATE}$ and puts a control token in $d'_{\Diamond}$ to initiate the unrolling of the BRCPN-EP.

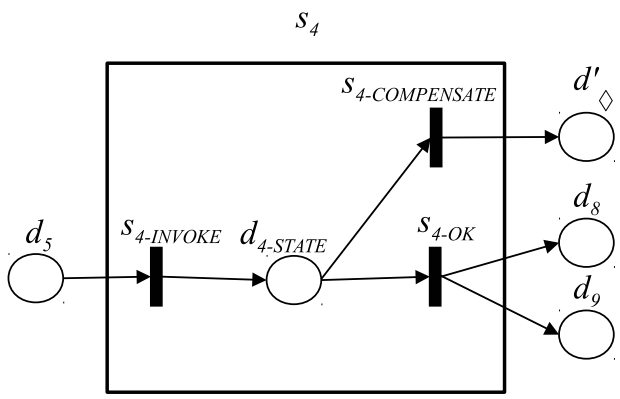


Figure 3.6: Internal Mechanisms of a Transition using Compensation.

3.1.2 Checkpointing

During the unrolling of a CPN-EP using the checkpointing mechanism, the execution of all services affected by a failure is skipped; the execution of services not affected by the failure continues as normal. To control checkpointing executions, we add a new type of control token to Def 3.1.1: $SKIP$. The $SKIP$ token controls the checkpointing execution flow of a CPN-EP, and it is visually represented by the roman letter "x". The execution of services may be skipped in two cases: if the service fails, or if the service is successor of a failed service. In both cases, the following skipping rules apply:

Definition 3.1.11 ***CPN-EP Skipping rules.*** *The skipping of a transition s for a marking M, defines a new marking M', such that all tokens are deleted from its input places:*

$$\forall d \in {}^{\bullet}s,\ M(d) \leftarrow \emptyset$$

and $SKIP$ control values are added to its output places:

$$\forall d \in (s^{\bullet}), M(d) \leftarrow M(d) \cup \{SKIP\}$$

These actions are also atomically executed. Differently from Def. 3.1.6, the skipping rules do not affect the states of transitions in BRCPN-EP.

If a transition is successor of a skipped transition, it is going to receive the $SKIP$ control token as part of its inputs; therefore, we define a *skippable* transition as follows:

Definition 3.1.12 ***Skippable transition.*** *A marking M enables a transition s for skipping if and only if all its input places contain tokens such that*

$$(\forall d \in {}^{\bullet}s,\ card(M(d)) = card({}^{\bullet}d)) \ \wedge\ (\exists d \in {}^{\bullet}s,\ \{SKIP\} \in M(d))$$

If there are skipped transitions at the end of a CPN-EP execution, we say that the execution is a *skipped* execution. Formally, a skipped execution is defined as follows:

Definition 3.1.13 ***Skipped Execution.*** *The execution of a CPN-EP is skipped if:*

$$\forall d \in CPN\text{-}EP_{OUT},\ M(d) = card({}^{\bullet}d) \wedge$$
$$\exists d \in CPN\text{-}EP_{OUT} \mid M(d) \in Bag(\{SKIP\})$$

When a composite service execution is skipped, the state of the CPN-EP is saved in persistent memory, and if partial output results exist, they are delivered to the user. We call the state of a skipped execution a CPN-EP snapshot CPN-$EP_{snapshot}$. This way, the composite service execution may be resumed later.

We illustrate the proposed checkpointing mechanism using the same example of Figure 3.4 where s_4 fails. Services s_1, s_2, and s_5 have been successfully executed. s_3 is fireable, and s_8 has already its input corresponding to d_9, but it is still waiting for the one corresponding to d_8. If the checkpointing option is available, the CPN-EP unrolling continues. The execution of s_4 is skipped (Def. 3.1.11), the $SKIP$ control token is set to the output places of s_4, d_8 and d_9, and s_7 and s_8 become skippable (Def. 3.1.12). Since s_3 was not affected by the failure of s_4, it continues its execution as if nothing happened (Figure 3.7 (b)).

Finally, Figure 3.7 (c) shows the state when the CPN-EP finishes its skipped execution. The output d_{10} was correctly produced, which is not the case for outputs d_{11} and d_{12}. Therefore, Def. 3.1.13 is satisfied since the three output places contain their necessary tokens, and at least one of them contains the $SKIP$ control value. In this case, both d_{11} and d_{12} contain the $SKIP$ control value.

Figure 3.7 (d) shows the part CPN-EP' of the CPN-EP that has to be resumed. We can see that the only services affected by the failure of s_4 were s_7 and s_8. Note that even though d_9 has the service s_4 as predecessor, it also forms part of the set of inputs of CPN-EP' since it was already produced by s_5. The complete sets of inputs and outputs of CPN-EP' are the following:

$$\text{CPN-}EP'_{IN} = \{d_5, d_9\}$$
$$\text{CPN-}EP'_{OUT} = \{d_{11,12}\}$$

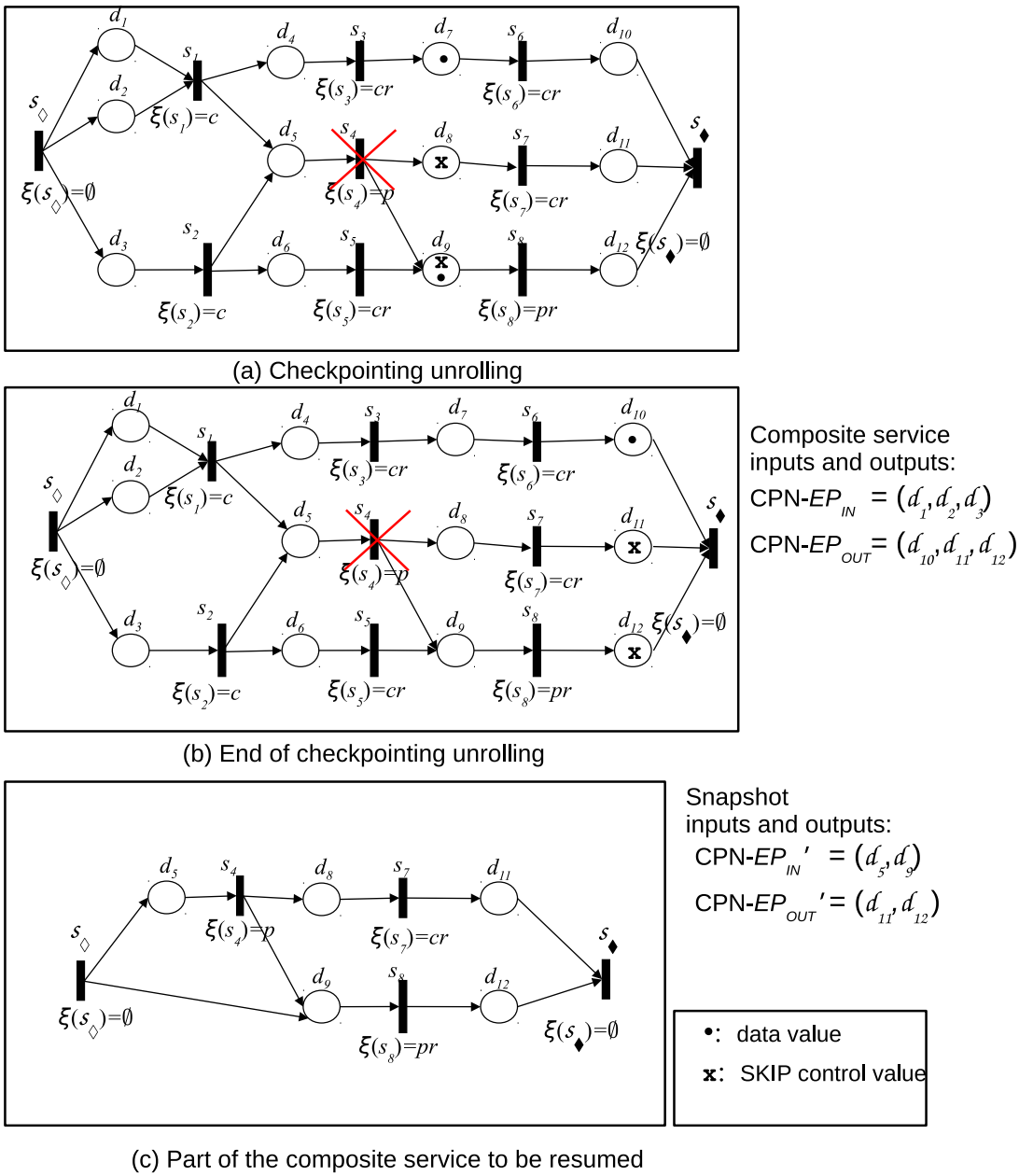


Figure 3.7: Checkpointing Example.

Note that a checkpointed composite service leaves the system in a temporarily inconsistent state; therefore, it must be resumed. A resumed composite service may either finish successfully, or fail again and be compensated. A user may not want to resume a checkpointed service if he obtained the outputs he wanted as part of the checkpointing partial outputs. In this case, the atomicity property is responsibility of that user.

To model the internal mechanism of checkpointing for a transition s_i, we add a transition $s_{i-CHECKPOINT}$ representing the checkpointing action of s_i. This transition is added to all services as follows:

$$\forall s_i \in S,\ S \leftarrow S \cup \{s_{i-CHECKPOINT}\}$$
$$\forall s_i \in S,\ F(d_{i-STATE}, s_{i-CHECKPOINT}) = 1$$
$$\forall d_i \in s_i^{\bullet},\ F(s_{i-CHECKPOINT}, d_i) = 1$$

When $s_{i-CHECKPOINT}$ is fired, it removes tokens from s_{i-OK} and sets skip control tokens to its corresponding output places:

$$M(d_{i-STATE}) \leftarrow \emptyset$$
$$\forall d \in s_{i-CHECKPOINT}{}^{\bullet},\ (M(d) \leftarrow M(d) \cup \{SKIP\})$$

A non-retriable service is internally composed by the transitions $s_{i-INVOKE}$, s_{i-OK}, $s_{i-COMPENSATE}$, and $s_{i-CHECKPOINT}$; a retriable service is internally composed by $s_{i-INVOKE}$, s_{i-OK}, $s_{i-RETRY}$, and $s_{i-CHECKPOINT}$.

Figure 3.8 illustrates the internal mechanisms of a transition using checkpointing. In this case, $s_{3-CHECKPOINT}$ removes the token from $d_{3-STATE}$ and puts a $SKIP$ control token in d_7 to initiate or continue the checkpointing process.

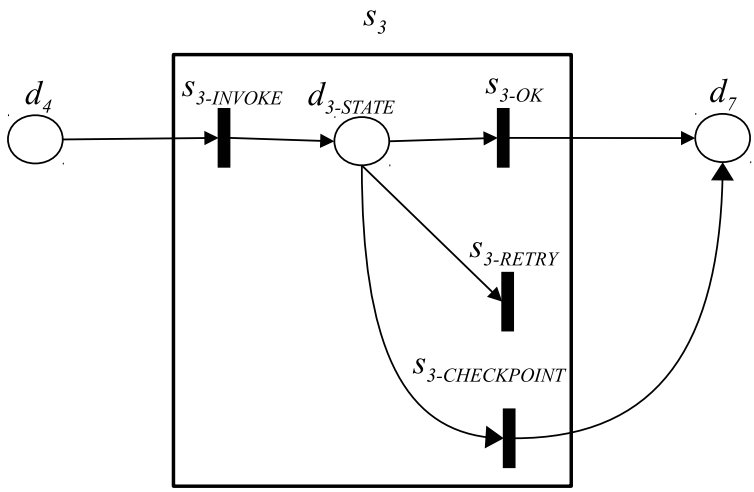


Figure 3.8: Internal Mechanisms of a Transition using Checkpointing.

3.1.3 Service Replacement

During the execution of composite services, if a failure occurs after many of its components services haven been successfully executed, a backward recovery process may lead to a significant resource wastage. Services with the *retriable* transactional property guarantee a successful execution after a finite number of invocations (Section 2.2.2); however, it is not always possible to provide a *retriable* composite service in which all its components are retriable to guarantee forward recovery. To avoid aborting the composite service execution and doing backward recovery, the replacement of a malfunctioning service is a forward recovery alternative. Therefore, when a service fails, if it is not retriable, we search for a replacement service to be invoked on behalf of the faulty one.

As we have explained in Section 2.1, services have already been discovery and classified according to their functionality; for example, using approaches such as [66] or [79]. Services are grouped in classes with the same functionality, but they can have different service descriptions, input and output attributes, transactional property, and QoS. Hence, we can define service functional equivalence according to the services input and output attributes. A service s is a functional replacement, denoted by $\equiv_F$, to another service s^*, if s^* can be invoked with at most the input attributes of s, and s^* produces at least the same output attributes produced by s. A functional replacement service is defined as follows:

Definition 3.1.14 ***Functional Replacement.*** *Let s and s^* be two services. We say that s^* is functional replacement of s, denoted as $s^* \equiv_F s$, if:*

$$\forall d^* \in {}^\bullet(s^*),\ \exists d \in {}^\bullet s \mid d \subseteq_{is-A} d^*,\ and$$
$$\forall d \in s^\bullet,\ \exists d^* \in (s^*)^\bullet \mid d^* \subseteq_{is-A} d$$

Figure 3.9 shows an example of different services in the same functional class. These services have different input and output attributes, transactional property, and though not showed in the example, may have different QoS. Regarding their inputs and outputs data types, suppose that they are associated with the example ontology described in the Figure 2.2 of Section 2.1.

By Def. 3.1.14 $s_1 \equiv_F s_3$ since $d_{13} \subseteq_{is-A} d_2$, but $s_3 \not\equiv_F s_1$ since $d_2 \not\subseteq_{is-A} d_{13}$. Similarly, $s_1 \equiv_F s_4$ since $d_3 \subseteq_{is-A} d_{11}$, but $s_4 \not\equiv_F s_1$. $s_3 \not\equiv_F s_4$ since $d_2 \not\subseteq_{is-A} d_{13}$

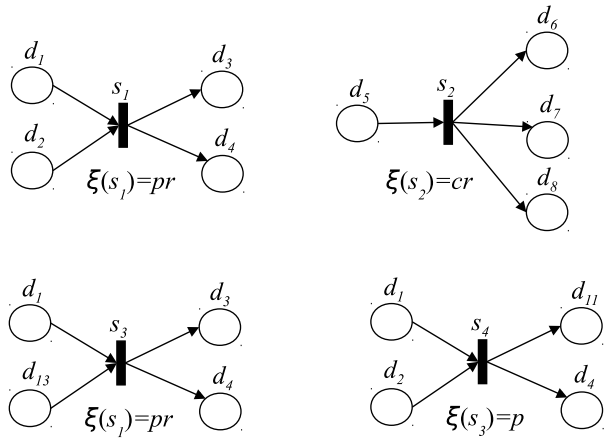


Figure 3.9: Functional Replacement Example.

and $s_4 \not\equiv_F s_3$ since $d_{11} \not\subseteq_{is-A} d_3$. Finally, s_2 cannot be replaced by any of the other services because s_1, s_2, s_3 require more inputs than s_2. s_2 can replace any of the other services since $d_1 \subseteq_{is-A} d_5$, and $d_6 \subseteq_{is-A} d_3$, $d_6 \subseteq_{is-A} d_{11}$ and $d_7 \subseteq_{is-A}$.

To guarantee the composite service global transactional property, also defined in Section 2.2.2, a service s can be replaced by another service s^*, if s^* can behave as s in the recovery process. Hence, if $TP(s) \in \{p, a\}$, in which case s only allows backward recovery, it can be replaced by any other service because all transactional properties allow backward recovery. A service with $TP(s) \in \{pr, ar\}$ can be replaced by any other retriable service (pr,ar,cr), because all of them allow forward recovery. A compensable service can be replaced by a service that also provides compensation; that is, c and cr services. A cr service can be only replaced by another cr service because it is the only transactional property allowing backward and forward recovery. We have said that to replace a service s with another service s^*, they must satisfy Def. 3.1.14. However, if s is compensable, its associated service s' must also be replaced in the BRCPN-EP (Def. 3.1.7) by the compensation service of s^*; that is, by a replacement compensation service s'^*.

Thus, a service s is transactional replacement of another service s^*, denoted by $\equiv_T$, if s is a functional replacement of s^* and their transactional property allow the replacement. Def. 3.1.15 shows the rules for a transactional replacement.

Definition 3.1.15 ***Transactional Replacement*** *Let s and s^* be two services, and s' and s'^* their corresponding compensation services if they exist. We say that s^* is transactional replacement of s, denoted as $s^* \equiv_T s$ if any of the following statements is true:*

- $(TP(s) \in \{p, a\} \wedge\ TP(s^*) \in \{p, pr, a, ar, c, cr\}) \wedge\ s^* \equiv_F s$
- $(TP(s) =\in \{pr, ar\} \wedge\ TP(s^*) \in \{pr, ar, cr\}) \wedge\ s^* \equiv_F s$
- $(TP(s) = c \wedge\ TP(s^*) \in \{c, cr\}) \wedge\ (s^* \equiv_F s \wedge\ s'^* \equiv_F s')$
- $(TP(s) = cr \wedge\ TP(s^*) \in \{cr\}) \wedge\ (s^* \equiv_F s \wedge\ s'^* \equiv_F s')$

In Figure 3.9, $s_1 \equiv_T s_3$ and $s_1 \equiv_T s_4$, but $s_4 \not\equiv_T s_1$. $s_2 \equiv_T s_1$, $s_2 \equiv_T s_3$, and $s_2 \equiv_T s_4$. s_3 satisfies the transactional property of s_1 and s_4 but s_3 do not satisfy the functional replacement definition for those services. s_4 cannot replace any of the services since s_1 and s_3 require the retriable property, and s_2 requires the compensable and retriable properties. The following definition presents the steps to take in case of service replacement:

Definition 3.1.16 ***Transactional Replacement Steps.***
Let CPN-EP $= (D, S, F, \xi)$ *the Colored Petri net allowing the execution of a composite service and BRCPN-EP* $= (D', S', F^{-1}, \zeta)$ *its corresponding backward recovery Colored Petri net. In case a service* $s \in S$ *fails, it can be replaced by another* s^**, if* $s \equiv_T s^*$*. Then, the following actions proceed:*

1) $S \leftarrow S \cup \{s^*\}$

2) $\forall d \in^\bullet (s^*),\ F(d, s^*) \leftarrow 1 \wedge \forall d \in^\bullet s,\ F(d, s) \leftarrow 0$*;*

3) $\forall d \in s^\bullet,\ F(s^*, d) \leftarrow 1,\ F(s, d) \leftarrow 0$*;*

4) $S \leftarrow S - \{s\}$*;*

5) if $TP(s) \in \{c, cr\}$*,* $s' \in S'$ *is replaced by* s'^* *in the BRCPN-EP, since* s'^* *compensates* s^**;*

6) $TP(s^*) \leftarrow TP(s)$*.*

When a substitution occurs, the faulty service s is removed from the CPN-EP, the new s^* is added, but the original CPN-EP structure remains intact. For compensable services, it is necessary a service replacement capable of maintaining the compensation control flow in the respective BRCPN-EP. In fact, when a compensable service is replaced, the corresponding compensation service must be also replaced by the new one in the BRCPN-EP.

The goal is to finish the execution with the same properties of the original composite service. Let us go back to Figure 3.4 which shows the state of the example composite service execution when s_4 fails. Figure 3.10 (a) shows the set of services providing the same functionality as s_4. Note that services in Figure 3.10 (a) have different inputs and outputs, and transactional properties. Regarding transactionality, s_4 can be replaced for any of the four services since it is pivot. We cannot replace s_4 by s_{23} because we do not have enough input data to invoke s_{23} and thus, it does not satisfy the functional replacement definition (Def. 3.1.14). Now, suppose that s_{20}, s_{21}, and s_{22} are all valid replacements for s_4; that is, for example:

$$d_5 \subseteq_{is-A} d_{20}$$
$$d_5 \subseteq_{is-A} d_{24}$$
$$d_5 \subseteq_{is-A} d_{28}$$

meaning that d_5 is the same concept or a subconcept of d_{20}, d_{24}, and d_{28}, and

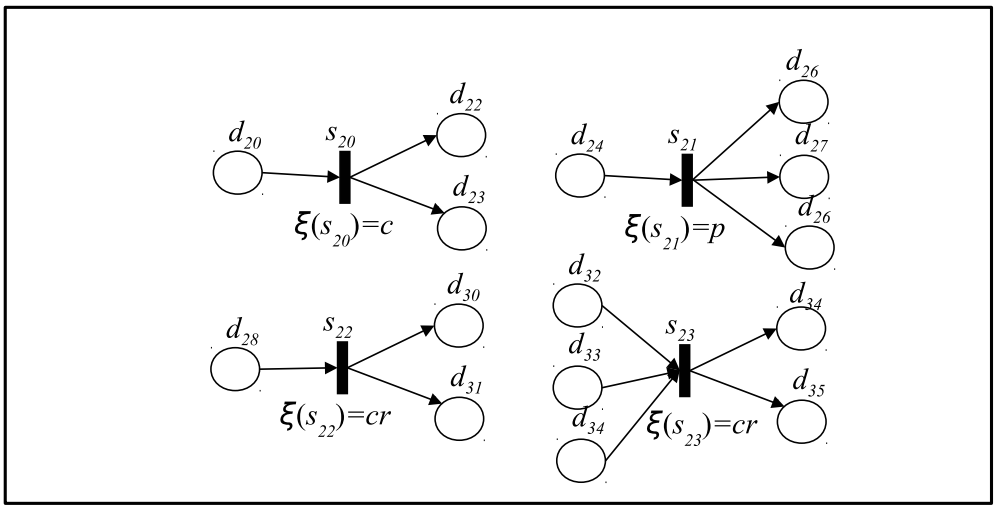
$$d_{22} \subseteq_{is-A} d_8$$
$$d_{26} \subseteq_{is-A} d_8$$
$$d_{30} \subseteq_{is-A} d_8$$

meaning that d_{22}, d_{26}, and d_{30} are the same concepts or subconcepts of d_8. d_{23}, d_{26}, d_{27}, and d_{31} are data not needed by the composite service of Figure 3.4. Finally, we suppose that s_{20}, s_{21}, and s_{22} are order by QoS:

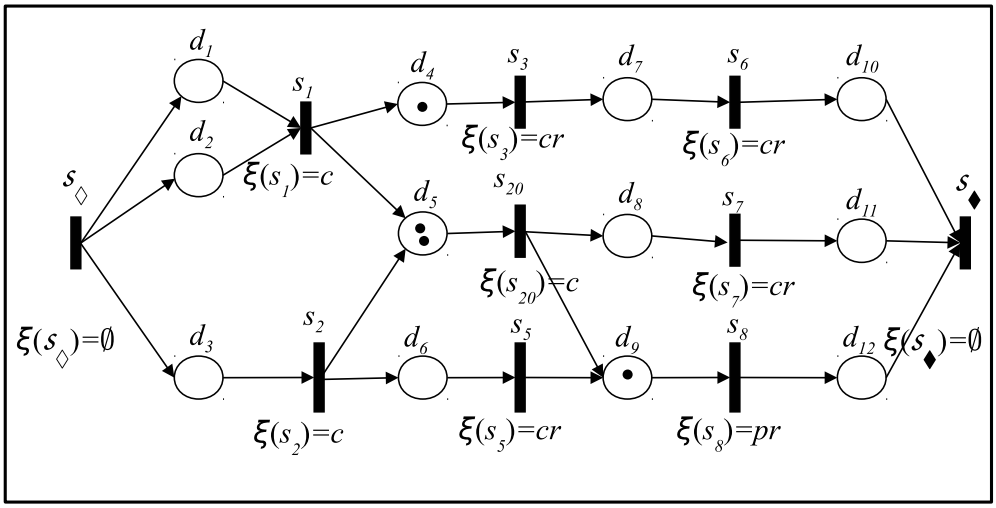
$$s_{20} \succ s_{22} \succ s_{21}$$

where s_{20} is the service with the best QoS, and s_{21} is the service with the worst QoS; therefore, s_{20} is chosen as replacement (Figure 3.10 (b)).

In case of failure of a service s, depending on its transactional property, the following actions can be executed:



(a) Services with the same functionality as s_4



(b) Replacement of s_4 by s_{20}

Figure 3.10: Service Replacement Example.

- if $TP(s)$ is retriable (pr,ar,cr), s is reinvoked until it finishes successfully (forward recovery);
- otherwise, another transactional replacement service, s^*, is selected to replace s, and the unrolling algorithm goes on (forward recovery);
- if there are no replacement services, a backward recovery is needed; i.e., all executed services must be compensated in the inverse order they were executed. For parallel services, the execution order does not matter.

When there exist several services candidates for replacing a faulty service s, the one with the best QoS is selected. The service replacement is done such that the replacement service locally optimizes the QoS. If several services have the same QoS, then the service replacement is chosen by transactional property according to Def. 3.1.15 and the following preference relation:

$$p = a \prec c \prec pr = ar \prec cr$$

since we prefer compensable over pivot/atomic services, and retriable services over non-retriable ones. Note that if more than one replacement is done for the same service, all of them take into account the original required transactional property, and not the transactional property of replaced service (Def. 3.1.16). This is to avoid unnecessary restrictions imposed by the transactional property of the replacement service. For example, the transactional property of the failed service in Figure 3.10 is p. Suppose that s_4 was replaced by another service s^* with transactional property c. However, if it turns out that s^* also fails, we have to perform service replacement again. s^* with $TP(s^*) = c$ can then be replaced by services with any transactional property, since the original requirement was p.

The internal mechanisms of service replacement are similar to the ones of service retry showed in Figure 3.3.

3.2 Framework Architecture

In this section, we present the overall architecture of our execution framework and a detailed explanation of its fault tolerance algorithms.

During the composite service execution there exist two basic variants of execution scenarios for component services. In a *sequential* scenario, services work on the result of previous services and cannot be invoked until previous services have finished. In *parallel* scenario, several services can be invoked simultaneously because they do not have data flow dependencies. The global transactional property of composite services is affected by these execution scenarios. Hence, it is mandatory to follow the execution flow defined by the CPN-EP (Def. 3.1.1) to ensure that sequential and parallel execution satisfy the global transactional property.

The execution of a composite service in our framework is managed by an Agent Coordinator and a collection (Γ) of software components called Service Agents (γ), organized in a three-level architecture. Figure 3.11 depicts the overall architecture of our framework. In the first level, the Agent Coordinator receives the composite service and its corresponding backward recovery graph, both represented by Colored Petri nets. These Colored Petri nets can be automatically or manually generated.

The Agent Coordinator launches in the second layer a Service Agent for each service in the composite service. Figure 3.11 shows an example containing the services:

$$S = \{s_1, s_4, s_6, s_7\}$$

Therefore, the following Service Agents are instantiated:

$$\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$$

where γ_1 is responsible for s_1, γ_2 for s_4, γ_3 for s_6, and γ_4 for s_7. Each Service Agent is responsible for the execution control of its service; that is, Service Agents:

- are responsible for the actual invocation of services;
- monitor the execution of their corresponding services;
- forward results to their peers to continue the execution flow;
- take fault tolerance actions in case of failure.

By distributing the responsibility of executing a composite service across several Service Agents, the logical model of our framework enables distributed execution and implementation independence. For example, this model can be implemented in a distributed memory environment supported by message passing, or in a shared memory platform supported by a distributed shared memory or tuplespace system.

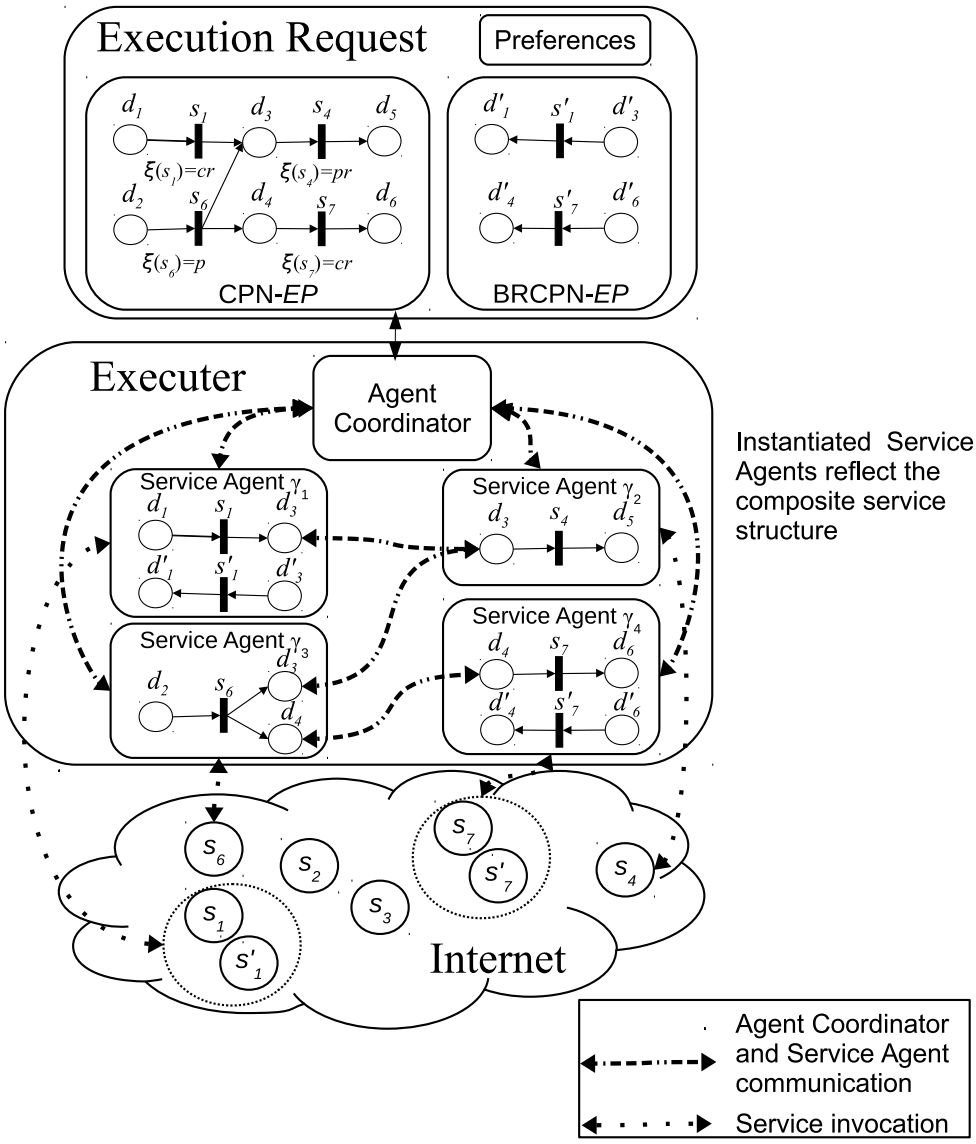


Figure 3.11: Execution System Architecture.

The idea is to place the Agent Coordinator and the Service Agents in different physical nodes with high availability and reliability; for example, in a cloud computing environment. These highly reliable nodes will be not the same as where the actual services are placed. Service Agents remotely invoke the actual component services. The knowledge required at runtime by each Service Agent (service descriptions, services predecessors and successors, and execution flow control) can be directly extracted from the Colored Petri nets in a shared memory implementation or sent by the Agent Coordinator in a distributed implementation.

3.2.1 Fault Tolerance Algorithms

This section presents the algorithms implementing the execution control and fault tolerance mechanisms for composite services. The whole execution process is divided in several phases, in which the Agent Coordinator and Service Agents participate. Table 3.2 shows implementation parameters and control messages. Control messages are special messages sent within the framework to perform the fault tolerant execution control of composite services.

	Type	*Description*
$CHECKPOINT_ENABLED$	Parameter	Enables the checkpointing option.
MAX_TRIES	Parameter	Maximum number of times a Service Agent can replace a service.
$COMPENSATE_READY$	Control Message	Prepares components for compensation.
$FINISH$	Control Message	Ends a Service Agents lifecycle.
$CHECKPOINT$	Control Message	Requests Service Agents to send their checkpointing data.
$CTRL_TOKEN$	Control Token	Control the execution flow.
$SKIP$	Control Token	Control the checkpointing unrolling process.

Table 3.2: Framework Parameters and Control Messages.

3.2.1.1 Initial phase

Whenever an Agent Coordinator receives a CPN-EP and its corresponding BRCPN-EP, it instantiates a Service Agent for each transition in CPN-EP (Algorithm 3.1 line 4). Service Agents are instantiated with the following information:

- predecessor Service Agents;
- successor Service Agents;
- the information related to its corresponding service.

This means that the Agent Coordinator sends the part of the CPN-EP that each Service Agent is interested in. Finally, it starts the CPN-EP unrolling by sending the attribute values in CPN-EP_{IN} to Service Agents responsible for the successors of the initial transition (Def. 3.1.3). In Algorithm 3.1, lines 1 to 8 describe these steps.

3.2.1.2 Service Invocation phase

Once a Service Agent has been instantiated, it waits until its corresponding service becomes fireable (Algorithm 3.2 line 2). Note that in the Service Agent context, γ_{IN} represents a dictionary containing the input data types d as keys, and their corresponding received data values and/or control tokens as dictionary values. When a Service Agent is waiting for its inputs, its corresponding service may become fireable for execution (Def. 3.1.5) or it may have become skippable (Def. 3.1.12). When a Service Agent receives all its needed inputs, it becomes fireable and it invokes its corresponding service (Algorithm 3.2 line 7). Upon the successfully completion of a service, its corresponding Service Agent sends the produced output values to its successor Service Agents. This step emulates the firing rules in CPN-EP. Note that all fireable services can be invoked in parallel for execution or for skipping. If a service fails during the execution (Algorithm 3.2 line 8), the Service Agents tries to perform forward recovery:

1) first, it verifies if its corresponding service is retriable. If it is, then the service is reinvoked;
2) if the service is not retriable, then the Service Agent tries to replace it by another service.
3) if the service is nor retriable and it does not have a replacement service, the Service Agent is left with two options: compensate or checkpoint. The default action is to compensate; checkpointing is chosen if the checkpointing option is enabled.

When a Service Agent has finished with the execution control of its corresponding service, it goes to the *final phase* (Algorithm 3.3).

3.2.1.3 Final phase

This phase is carried out by the Agent Coordinator and Service Agents (Algorithms 3.1 line 9 and Algorithm 3.3, respectively). After it has instantiated all Service Agents and initiated the composite service execution, the Agent Coordinator goes to this phase and waits for the execution termination or for a message indicating that it is necessary to compensate. In case the composite service execution finishes successfully, the Agent Coordinator receives all the composite

service outputs and terminates all Service Agents by sending the *FINISH* message (Algorithm 3.1 line 12). Then, it recalculates the composite service QoS, and returns the values in CPN-EP_{OUT} to the user. When a Service Agents receives the *FINISH* message, it terminates its execution.

In case compensation is needed, the Agent Coordinator receives a *COMPENSATE_READY* message, and it executes the compensation phase. If a Service Agent receives a *COMPENSATE_READY* message, it also launches its compensation protocol.

Finally, if the Agent Coordinator receives an $SKIP$ control token for at least one of the outputs in CPN-EP_{OUT}, it executes the *checkpointing phase* (Algorithm 3.1 line 14).

3.2.1.4 Replacing phase

If a failure occurs during a service execution, the corresponding Service Agent checks if the service is retriable. If it is not retriable, it applies service replacement (Algorithm 3.4). Services can be replaced if there exist candidate services and while a maximum number of replacements has not been reached (Algorithm 3.4 line 2). From the set of candidate services that can replace the failed service functionally and transactionally, the Service Agents selects the best one regarding QoS.

Lines from 4 to 6 show the necessary steps to replace the service in a CPN-EP, including the service replacement in the BRCPN-EP (Def. 3.1.7) if the replaced service is compensable.

3.2.1.5 Compensation phase

If forward recovery is not possible and checkpointing is not enabled, compensation is chosen to leave the system in a consistent state. The Service Agent responsible for the faulty service informs the Agent Coordinator about the failure (Algorithm 3.2 line 10), then it goes to the compensation phase (Algorithm 3.5 line 3).

The Agent Coordinator sends a message *COMPENSATE_READY* to all Service Agents (Algorithm 3.5 line 2) and starts the compensation process following

an unrolling algorithm over the BRCPN-EP. Once all the Service Agents receive the message $COMPENSATE_READY$, they apply the firing rules in BRCPN-EP to follow the compensation process. The compensation steps for Service Agents are showed in Algorithm 3.5 line 2.

3.2.1.6 Checkpointing phase

This phase is carried out by the Agent Coordinator and the Service Agents which cannot invoke their corresponding services, because they are in the path of a failure, or their corresponding service failed.

In Algorithm 3.2 line 3, the Service Agent sends a $SKIP$ message to its successors following the skipping rules (Def 3.1.11) and skippable transition (Def. 3.1.12) definitions. If its corresponding service fails (Algorithm 3.2 line 8) and the checkpointing option is enabled, the Service Agent also triggers the skipping rules, sending a $SKIP$ message to its successors. Service Agents which corresponding service fails are the ones that trigger the checkpointing process in Algorithm 3.2 line 9.

Finally, a Service Agent which corresponding service execution was skipped waits until it receives a $CHECKPOINT$ message (Algorithm 3.3 line 1). When a Service Agent receives a $CHECKPOINT$ message, it sends its information to the Agent Coordinator. This information consists of data values and/or control tokens received as input, the information of its corresponding service, and produced outputs if they exist.

In Algorithm 3.1 line 14, in case of a skipped execution (Def. 3.1.13), the Agent Coordinator saves the produced outputs CPN-EP_{OUT} of the composite service, and collects the information of Service Agents. Then, it returns a CPN-$EP_{snapshot}$ containing the part of the CPN-EP that could not be executed, the needed inputs to resume it, and the produced composite service outputs.

3.2.1.7 Resume phase

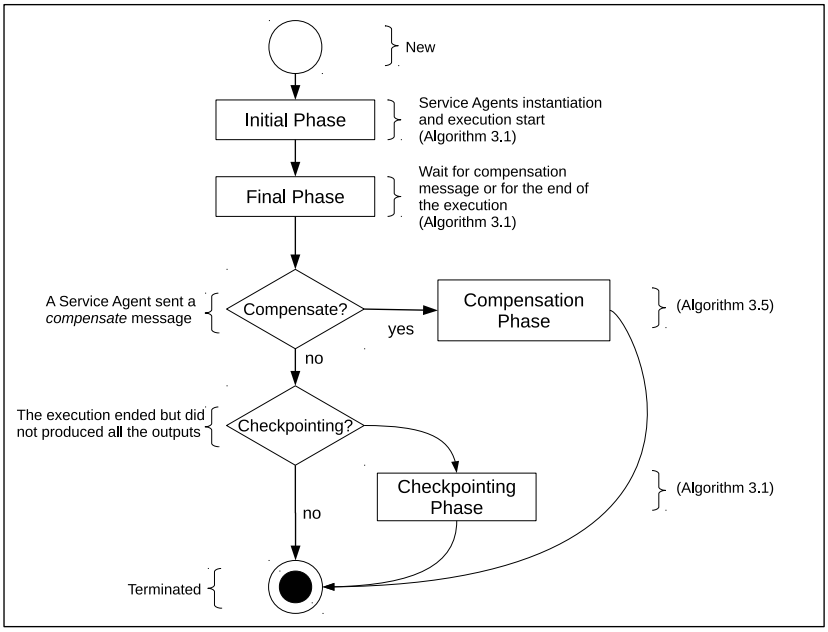
There is no difference between a resumed execution of a checkpointed CPN-EP and a non-checkpointed CPN-EP execution. In this phase, both Agent Coordinator and Service agents execute a checkpointed CPN-EP by using the previously described algorithms as for a normal execution.

Figures 3.12 depicts the flow diagrams showing the phases previously described for the Agent Coordinator and Service Agents and their relation with the presented algorithms.

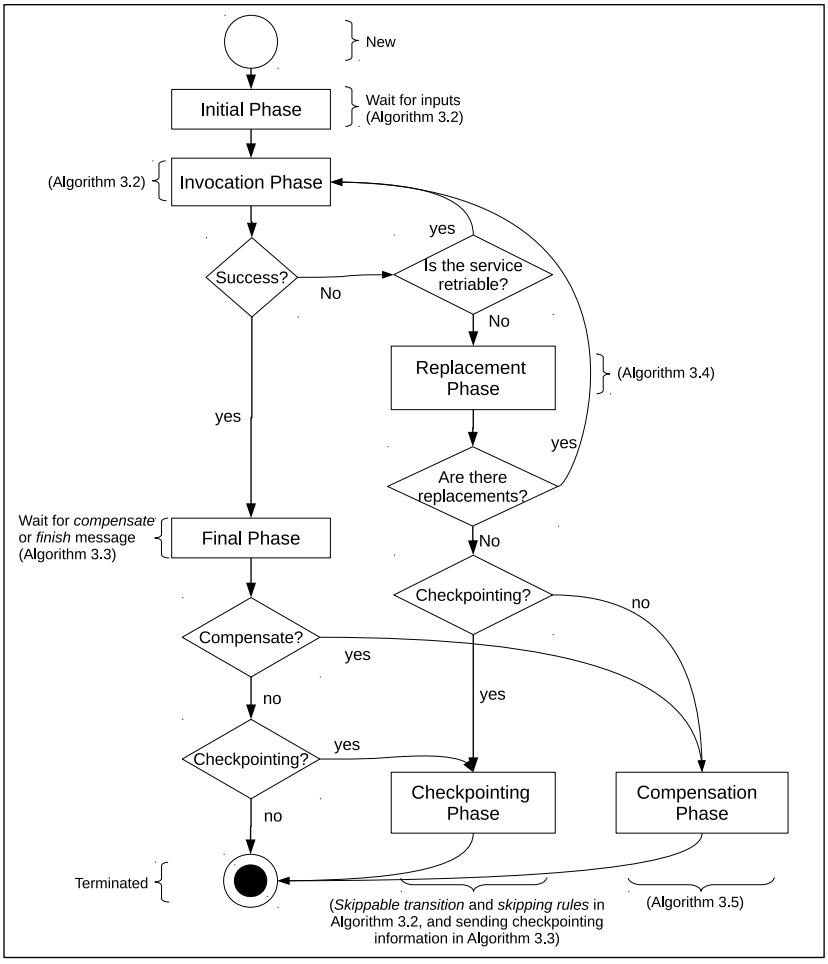
Algorithm 3.1 Agent Coordinator Algorithm.

Input: CPN-$EP = (D, S, F, \xi)$, a Colored Petri net representing a composite service
Input: BRCPN-$EP = (D', S', F^{-1}, \zeta)$, a Colored Petri net representing the compensation flow of CPN-EP
Output: CPN-EP_{OUT}: composite service outputs

1 **Initial phase**:
begin
  /* The state of all transitions in BRCPN-EP is *initial* */
2 $\forall s' \in S'$, $\zeta(s') \leftarrow I$;
3 **repeat**
4 Instantiate a Service Agent γ responsible for s;
5 Send predecessors reference ${}^{\bullet}({}^{\bullet}\gamma)$ to γ ;
6 Send successors reference $(\gamma^{\bullet})^{\bullet}$ to γ;
  until $\forall s \in S \mid (s \neq s_{\Diamond}) \wedge (s \neq s_{\blacklozenge})$;
7 Fire $\gamma_{\Diamond}$ with the values of CPN-EP_{IN};
8 **Execute Final phase** (line 9);
end
9 **Final phase**:
begin
10 **repeat**
  Wait for *message* from ${}^{\bullet}({}^{\bullet}\gamma_{\blacklozenge})$;
  if $message = COMPENSATE_READY$ **then**
  Execute Compensation phase; (Algorithm 3.5 line 1)
  Execute Final phase (line 9);
  else
  Put received output value in CPN-EP_{OUT};
  end
  until $\forall d \in$ *CPN-*EP_{OUT} , $M(d) = card({}^{\bullet}d)$;
  /*$\gamma_{\blacklozenge}$ is fireable*/
11 **if** $\exists d \in$ *CPN-*$EP_{OUT} \mid M(d) \in Bag(SKIP)$ **then**
  //Skipped execution Def. 3.1.13
  Execute Checkpointing phase (line 14);
  end
  else
12 Send $FINISH$ message to ${}^{\bullet}({}^{\bullet}\gamma_{\blacklozenge})$;
13 **Return** CPN-EP_{OUT};
  end
end
14 **Checkpointing phase**:
begin
  Save received data values of CPN-EP_{OUT};
  Send $CHECKPOINT$ message to Service Agents;
  Wait for responses from Service Agents;
  Build CPN-$EP_{snapshot}$;
  Return CPN-$EP_{snapshot}$;
end



(a) Agent Coordinator Algorithm



(b) Service Agent Algorithm

Figure 3.12: Agent Coordinator and Service Agent Flowcharts.

Algorithm 3.2 Service Agent Algorithm: Initial and Invocation Phases.

```
   Input: •(•γ), predecessors Service Agents
   Input: (γ•)•, successors Service Agent of γ
   Input: s: the corresponding service
   Input: s≡: equivalent services of γ
   Input: 𝒫: preferences //if checkpointing is enabled or not, and maximum number of replacements
   Output: ∅
1  Invocation phase:
   begin
       ∀d ∈ •γ, •γ_IN.put(d, ∅); //γ_IN is a dictionary with keys d with multiple values per key
2      repeat
           Wait for message from •(•γ) or Agent Coordinator;
           if message = COMPENSATE_READY then
               Execute Compensation phase (Algorithm 3.5 line 3);
               Return;
           else
               •γ_IN.put(message.d, message.d_value);
           end
       until ∀d ∈ •γ_IN, card(M(d)) = card(•d);
       /* all the predecessor transitions have finished */
3      if SKIP ∈ γ_IN then
           /*Skippable transition (Def. 3.1.12) */
           Send SKIP to (γ•)• ;
           Return;
       end
       /*Fireable transition (Def. 3.1.5) */
4      success ← false;
       cantry ← true;
       tries ← 0;
5      ζ(s′) ← R;
6      repeat
7          Invoke s;
           if (s fails) then
               if ¬TP(s) ∈ {pr, ar, cr} then
                   if s≡ ≠ ∅ ∧ tries < 𝒫.MAX_TRIES then
                       if ¬TP(s) ∈ {pr, ar, cr} then
                           Execute Replacing phase (Algorithm 3.4);
                           tries + +;
                       end
                   else
                       cantry ← false;
                   end
               end
           else
               ζ(s′) ← E;
               Send s outputs to (γ•)•;
               success ← true;
           end
       until success ∨ ¬cantry;
8      if ¬success then
9          if 𝒫.CHECKPOINT_ENABLED then
               Send SKIP to (γ•)•;
           else
10             Send COMPENSATE_READY message to Agent Coordinator;
               ζ(s′) ← C ;
               Execute Compensation phase (Algorithm 3.5 line 3)
           end
       else
           Execute Final phase (Algorithm 3.3)
       end
   end
```

Algorithm 3.3 Service Agent Algorithm: Final Phase.

```
1  Final phase:
   Input: ∅
   Output: ∅
   begin
2      Wait for message;
       if message = FINISH then
           Send FINISH message to •(•γ);
           Return;
       else
           if message = CHECKPOINT then
               //received data values and/or control tokens, and produced outputs
               Send Service Agent γ information to Agent Coordinator ;
3              Return;
           else
               if message = COMPENSATE_READY then
                   Execute Compensation phase (Algorithm 3.5 line 3)
                   Return;
               end
           end
       end
   end
```

Algorithm 3.4 Service Agent Algorithm: Replacing Phase.

```
1  Replacement phase:
   Input: s≡: equivalent services of s
   Output: s*: a service if s≡ ≠ ∅
   begin
2      if s≡ ≠ ∅ then
3          Select s* ∈ s≡| [ ∀s'' ∈ s≡, (QoS(s*) ≥ QoS(s'')) ];
4          S ← S ∪ {s*};
           ∀d ∈ •(s*), F(d, s*) ← 1∧ ∀d ∈ •s, F(d, s) ← 0;
5          ∀d ∈ s•, F(s*, d) ← 1, F(s, d) ← 0;
           S ← S − {s};
           s≡ ← s≡ − {s*};
           TP(s*) ← TP(s);
           if TP(s) ∈ {c, cr} then
               s' ∈ S' is replaced by s'* in the corresponding BRCPN-EP;
               /*it compensates s**/
           end
6          Return s*;
       else
7          Return ∅;
       end
   end
```

Algorithm 3.5 Agent Cordinator and Service Agent: Compensation Phase.

```
begin
1   Agent Coordinator:
    begin
        /*Mark the BRCPN-EP with the Initial Marking*/
2       Send COMPENSATE_READY to all Service Agents γ ∈ Γ;
        Send CTRL_TOKEN to •(•γ♦);
        Wait for CTRL_TOKEN from ((γ◊)•)•;
        Return ERROR;
    end
3   Service Agents:
    begin
        /* s′: compensation service of the corresponding service s*/
        if ζ(s′) = A ∨ ζ(s′) = C then
            Send CTRL_TOKEN to •(•γ);
        else
            repeat
                Wait for CTRL_TOKEN from (γ•)•;
                Set CTRL_TOKEN the corresponding input place d′ ∈• s′;
            until (∀d ∈ γ•, card(M(d)) = card(d•);
            /*γ may now fire the compensation service s′*/
            if ζ(s′) = I then
                ζ(s′) ← A
            end
            if ζ(s′) = R then
                Wait s finishes;
                Invoke s′;
                ζ(s′) ← C;
            end
            if ζ(s′) = E then
                Invoke s′;
                ζ(s′) ← C;
            end
            Send CTRL_TOKEN to •(•γ);
        end
        /*The Service Agent finishes */
        Return;
    end
end
```

3.3 Conclusions

In this chapter, we have formalized the reliable execution of composite services using Colored Petri nets. Our approach ensures the correct execution of composite services and provides fault tolerance if needed. The execution model is distributed, can be implemented in distributed or share memory systems, is independent of the implementation of services, and is transparent to users and developers. We base our fault tolerance on transactional properties to provide a deep-seated notion of what a correct state of a composite service is. The main considered recovery mechanisms are: forward recovery by retrying or replacing the faulty service, and backward recovery based on an unrolling process over a Colored Petri net representing the compensation flow of the faulty composite service.

Additionally, we have presented a checkpointing mechanism that provides an alternative to the service retry, service substitution, and compensation mechanisms. It allows to delay the execution of the faulty part of a composite service, while continuing the execution of the part not affected by the failure. Also, the checkpointing mechanism allows users to receive partial answers as soon as they are produced and provides the option of resuming the composite service without losing the work previously done, and without affecting the original transactional property. We also pointed out that a checkpointed composite service leaves the system in a temporarily inconsistent state; therefore, it must be resumed. A resumed composite service may either finish successfully, or fail again and be compensated.

As hypothetical limitations we can point out the following: the framework does not take into account QoS to make decisions, decisions are based on transactional properties and are only taken as a reaction to failures; it may be difficult to do a sound experimental evaluation due to the lack of testbeds for the execution of composite services under unreliable environments; and, it may be also difficult to deploy our system in the real-world due to the lack of interoperability, integration, and automation of inter-organization services.

In the next chapter, we present a knowledge-based approach for self-healing composite services. This approach extends the framework based on transactional properties presented in this chapter to provide smarter knowledge-based decisions.

Chapter 4

Knowledge-based Service Agents

Contents

4.1 Motivation **72**

4.2 A High-level Definition of Self-healing Composite Services **74**

4.3 Knowledge-Based Service Agents **79**

4.3.1 Self-awareness Knowledge 82

4.3.2 Context-awareness Knowledge 83

4.4 Knowledge Base **96**

4.4.1 QoS State Deduction 99

4.4.2 Self-healing State Deduction 100

4.4.3 Action Deduction 101

4.5 QoS Manager for Summation/Product QoS Criteria **105**

4.6 Algorithms **108**

4.7 Conclusions **111**

In the previous chapters, we have seen that different situations may cause a component service to fail during the execution of a composite service. However, a fault tolerant composite service is the one that, upon a service failure, ends up the whole composite service execution successfully using forward recovery techniques, or leaves the execution in a safe state using backward recovery. Examples of forward recovery techniques are service retrying and service replacement; while roll-back and compensation are examples of backward recovery techniques. In

this sense, reliable execution of composite services becomes a key mechanism to cope with challenges of open-world applications in dynamic, changing, and untrusted operating environments. The reliable execution of composite services ensures the consistent state of the whole system, even in presence of failures [98].

In this context, failures during of composite service executions can be repaired by backward or forward recovery processes. Backward recovery implies to undo the work done before the failure, and go back to a consistent state close to the one the system had before the composite service execution Forward recovery tries to repair the failure to continue the execution and finish it successfully. In Chapter 3, we have presented our backward recovery approach by compensation, and forward recovery by retrying. Both approaches are based on the transactional properties model for composite services explained in Section 2.2.2. In Section 3.1.3, we have presented our approach for forward recovery based on service replacement in case a failed service cannot be retried.

For some users, partial responses may have sense. Also, encountered failures may be temporary. Checkpointing techniques may be implemented to survive service failures by executing the part of the composite service not affected by those failures. Checkpointing may be implemented as a recovery technique independent of transactional properties [85]. As an alternative to backward and forward recovery, we have presented an approach for checkpointing in Section 3.1.2. In our approach, the execution of the faulty part of a composite service is delayed and put on stand-by to be resumed later. The part of the composite service not affected by failures continues its execution as if nothing happened.

4.1 Motivation

Because services can be created and updated on-the-fly, the execution system needs to dynamically detect changes during run-time, and adapt the execution to the availability of existing services. The highly dynamic nature of Internet and the compositional nature of services make the above static fault tolerance strategies unpractical in real-world environments where users not only care about system consistency, but also about QoS. This is why is necessary to create more sophisticated composite service execution systems capable of making smarter decisions at runtime. In fact, such a composite service execution system should take into account:

1) the composite service execution state at the moment of a failure; for example, how many component services have been successfully executed and how many have not been invoked;

2) the environment state; for example, network connectivity;

3) the impact of the recovery strategy in the composite service QoS.

In these scenarios some questions emerge to decide which recovery strategy is the best in terms of its impact on the composite services QoS: are all recovery techniques equally practical, effective, and efficient? When is it better to apply backward (or forward) recovery? Is it replication the best strategy? These unpredictable characteristics of SOA environments provide a challenge for optimal fault tolerance strategy selection. There is an urgent need for more general and smarter fault tolerance strategies, which are context-information aware and can be dynamically and automatically reconfigured for meeting different user requirements and adapting to changing environments. Hence, it is important to define a dynamic fault tolerant strategy which takes into account that kind of information to choose the most appropriate recovery strategy.

In the model presented in Chapter 3, the recovery decisions are taken based solely on the transactional capabilities of services; for example, according to the diagram of Figure 4.1, when a service fails, the decision making process is as follows:

1) if the failed service is retriable, then the recovery mechanism is forward recovery by retrying;

2) if it is not retriable and has a replacement, then the recovery mechanism is forward recovery by service replacement;

3) if it is not retriable and does not have replacements, the recovery mechanism is backward recovery by compensating unless the checkpointing option is enabled.

We call this described decision making process a *static* strategy selection. It is *static* since the possible recovery actions are predefined before the composite service execution starts; that is, they only depend on the composite service Colored Petri net structure, transactionality, and the availability of service replacements. From this static selection, we draw the following hypothetical issues:

1) *QoS degradation*: the execution time of the whole composition can be degraded due to the additional time required to perform retries and substitutions; execution time or any other QoS criteria can be degraded due to the QoS of a replacement service.

2) *Lost work*: compensating a composite service execution can lead to the loss of important work already done and resource wastage, such as waited time, payed services execution, used computational power, or generated user outputs.

3) *Checkpointing too early*: it is the opposite case of the previous point. It may not be worth it to checkpoint composite services executions that produced none or a just few user outputs, and executed a small percentage of it component services.

Moreover, QoS degradation can also occur during failure-free composite service executions due to dynamic nature of the execution environment and the component services.

Focused on the needs presented in this section, we present a new approach to deal with the limitations of static automatic fault tolerance selection for composite service executions. We start by giving a high-level definition of self-healing composite services. This definition allows the further understanding of composite service states, and how to take the necessary actions in case of failures or to maintain QoS.

4.2 A High-level Definition of Self-healing Composite Services

Before defining a self-healing composite service, we propose the definition of the following four states for composite services (Figure 4.2): "created", "running", "waiting", and "terminated". In the "created" state, the composite service has never been executed, while in the "terminated" state, the composite service has been executed either successfully or with failures. In the "waiting" state the composite service has been stopped to resume it later using the checkpointing technique. We define the self-healing states of composite services from the states of Figure 4.2 in the remaining of this section.

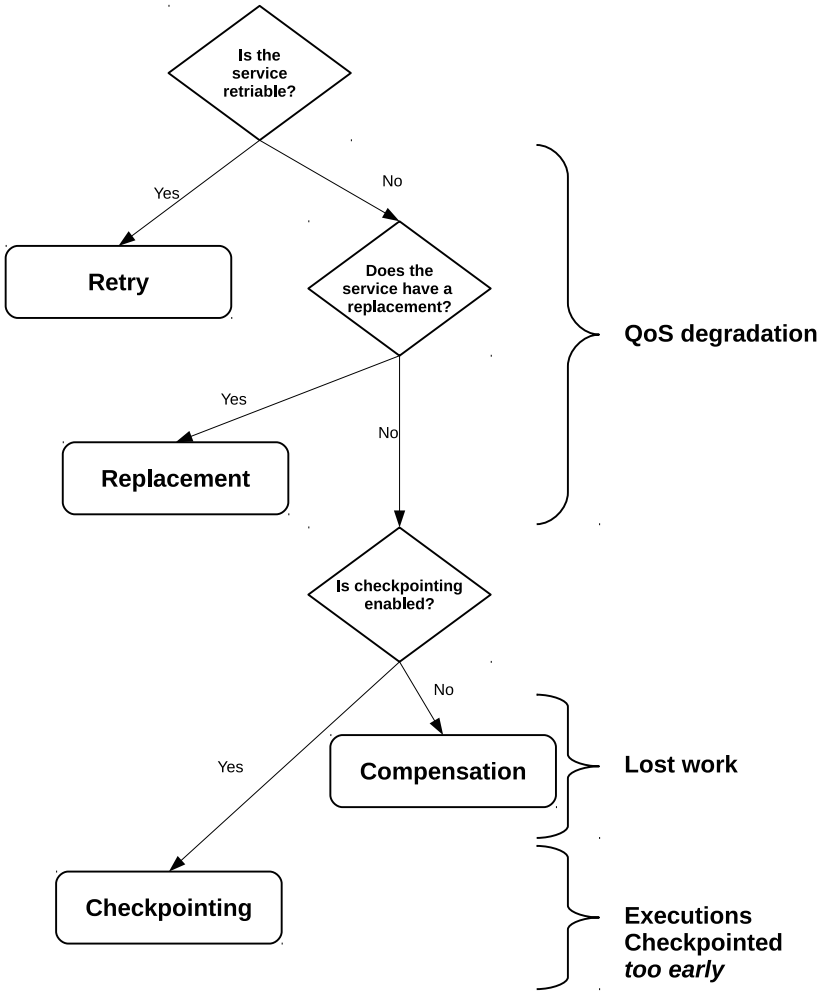


Figure 4.1: The Problems of the Static Selection of Fault Tolerance Strategies.

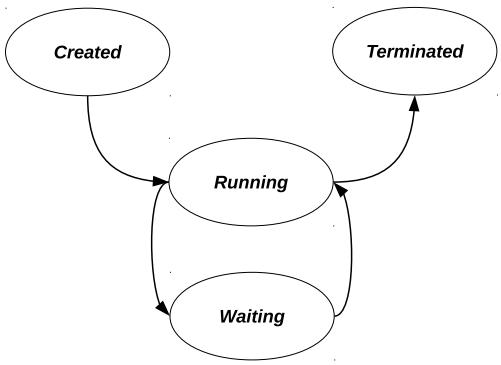


Figure 4.2: Execution States of a Composite Service.

In Section 2.3, we proposed an introduction to self-healing systems where Figure 2.6 on page 25 illustrates the three self-healing states: "normal state", "degraded state", and "broken state". We also stated the importance of distinguishing between these three self-healing states to make appropriate decisions regarding recovery strategies. In the approach presented in this chapter, the composite service QoS is a crucial indicator of the system behavior and health. We consider the following QoS notions:

1) **Expected QoS:** it refers to the estimated composite service QoS; for example, composite service execution time, price, and availability (Section 2.1).

2) **Acceptable QoS:** it refers to the degree to which the expected QoS can be degraded; for example, a user may be ready to wait more time or pay a higher price than expected for a composite service execution.

With the help of Figure 4.2, and using the notions of *expected* and *acceptable* QoS, we propose a high-level definition of the "normal", "degraded", and "broken" self-healing states for composite services. A composite service is in the **normal state** (Figure 4.3 (a)) if:

- the composite service is in the "created" state; therefore, the system is in a consistent state;
- the composite service is in the "terminated" state and it left the system in a consistent state by the means of fault tolerance techniques or failure free executions;
- the composite service is in the "running" state, there are no failures, and the QoS remains within the expected values. Preventive measures such as replication may be applied.

A composite service is in the **degraded state** (Figure 4.3 (b)) if:

- during its execution, the QoS is degraded but still within its acceptable values;
- during its execution, some failures occur without affecting the global composite service expected QoS;

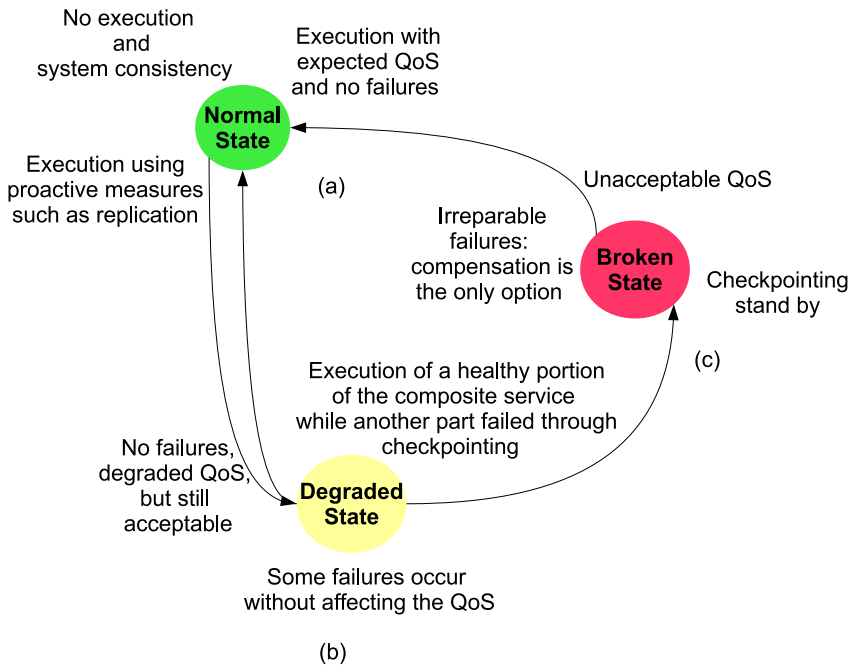


Figure 4.3: Self-healing States for Composite Services.

- during the checkpointing process; that is, a portion of the composite service cannot be executed due to a failure, while a portion of the composite service not affected by that failure continues its normal execution.

A composite service is in the **broken state** (Figure 4.3 (c)) if:

- during its execution, it has become unhealthy due to unacceptable QoS degradation;
- after a checkpointing process finished; that is, during checkpointing stand by;
- when an irreparable failure occurs and the only option is compensation.

In Section 3.2 we presented our composite service execution framework where Service Agents were responsible for the execution control of component services. In this chapter, we go deeper into the concept of Service Agents, and focus on what they know and what they can do. The main goal of Service Agents is to be autonomous components capable of making decisions by themselves. Taking decisions locally does not mean that components neglect their surroundings; on

the contrary, a Service Agent takes decisions based on the following three main observations:

1) what is expected to happen and what really happened before the invocation of its corresponding service;
2) what is expected to happen and what really happened during the invocation of its corresponding service;
3) what remains to happen after the invocation of its corresponding service.

These are three key notions that comprise the Service Agent knowledge for building self-healing composite services. Based on this, our design considers a self-healing loop per Service Agent to perform the detection, diagnosing, and recovering tasks in a decentralized fashion.

The "detection" component (Figure 4.4 (a)) takes into account one external and two internal data sources. The external information regards the expected QoS; for example, the user can allow a certain QoS degradation. The internal information refers to the QoS degradation of component services (e.g., negative variations in execution time and price), and to services failures, which is also a special case of QoS degradation.

The "diagnosis" component (Figure 4.4 (b)) analyzes the problem and does the triage of the composite service state. The three possible diagnosis correspond to the three states of a self-healing system: normal; degraded; and broken. The choice of the recovery mechanism is influenced by available options (e.g., replacement services, transactional properties, etc), and user preferences (e.g., expected QoS, checkpointing, etc).

The "recovery" component (Figure 4.4 (c)) is in charge of applying the selected fault tolerance mechanisms: backward recovery through compensation; forward recovery through service retry or substitution; prevention through replication; or delaying the execution through checkpointing.

The next section provides a deeper view on the Service Agent architecture and the formal definitions of the Service Agent knowledge model.

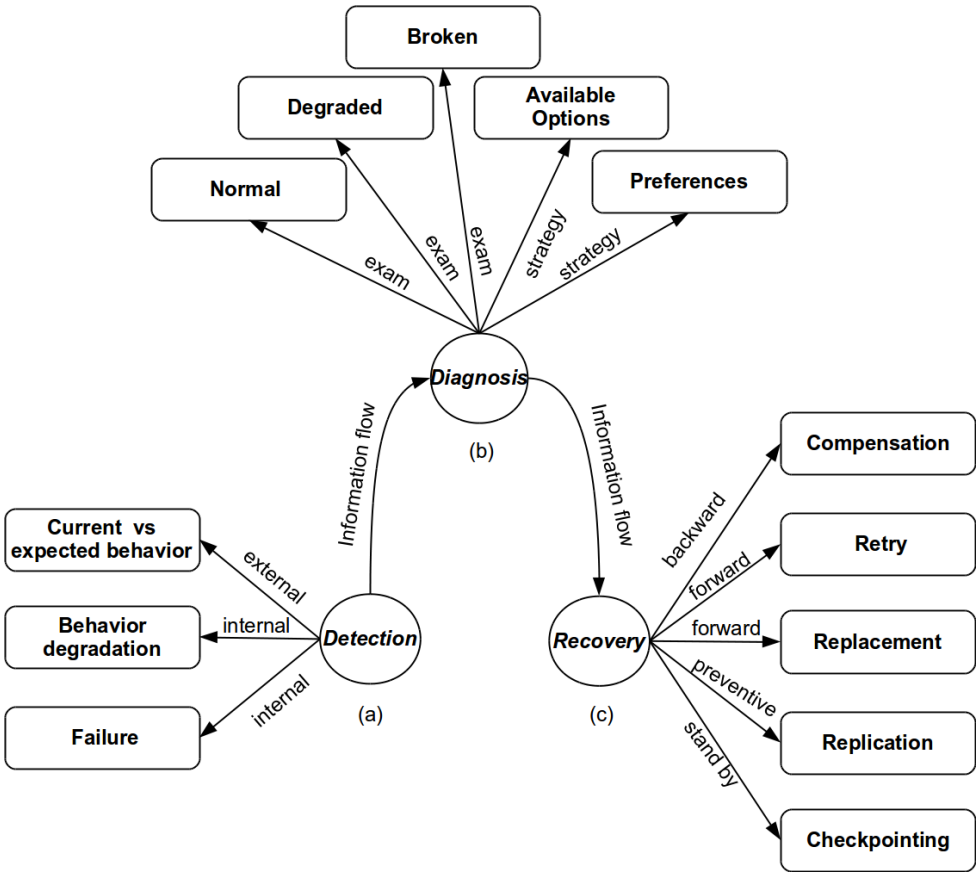


Figure 4.4: Service Agent Self-healing Loop.

4.3 Knowledge-Based Service Agents

We first introduced Service Agents as part of our execution framework in Section 3.2. As a reminder, they are software components in charge of the execution control of a service participating in a composite service execution. A Service Agent waits for inputs, invokes its corresponding service, implements fault tolerance mechanisms if necessary, and sends produced outputs to other Service Agents. Regarding the notation, a Service Agent is denoted by γ, and Γ is the set of all Service Agents participating in the same composite service execution. In this section, we elaborate on the description of the Service Agent architecture. The idea is to enhance Service Agents with knowledge-based decision making capabilities.

Figure 4.5 shows the architecture of Service Agents. The main components are the Service Agent API, the autonomic component, and the knowledge base:

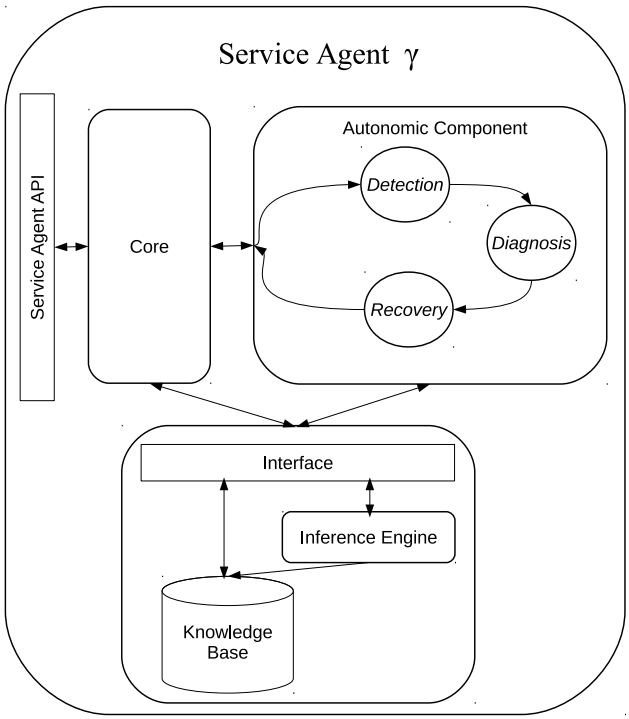


Figure 4.5: Service Agent Architecture.

- Service Agent API: Service Agents are capable of sharing what they know with other components through the Service Agent API. A Service Agent is capable of answering questions regarding its current status; for example, if it executed its corresponding service successfully or not. Service Agents may also be notified of context changes, may receive input data, execution control messages, etc, through their API. In summary, Service Agents provide two machine friendly interfaces: one to be *told* information; another to be *asked*.

- Core: it contains the basic execution control elements of Service Agents; that is, the algorithms presented in Section 3.2.

- Autonomic Component: it detects degradations on the composite service behavior, selects an appropriate action, and applies it.

- Knowledge Base: contains information about the Service Agent itself, its corresponding service, and the composite service execution context. It also contains a set of rules to transition between self-healing states, and a set of rules to deduce actions to take.

In the next section, we present and classify the knowledge required by a Service

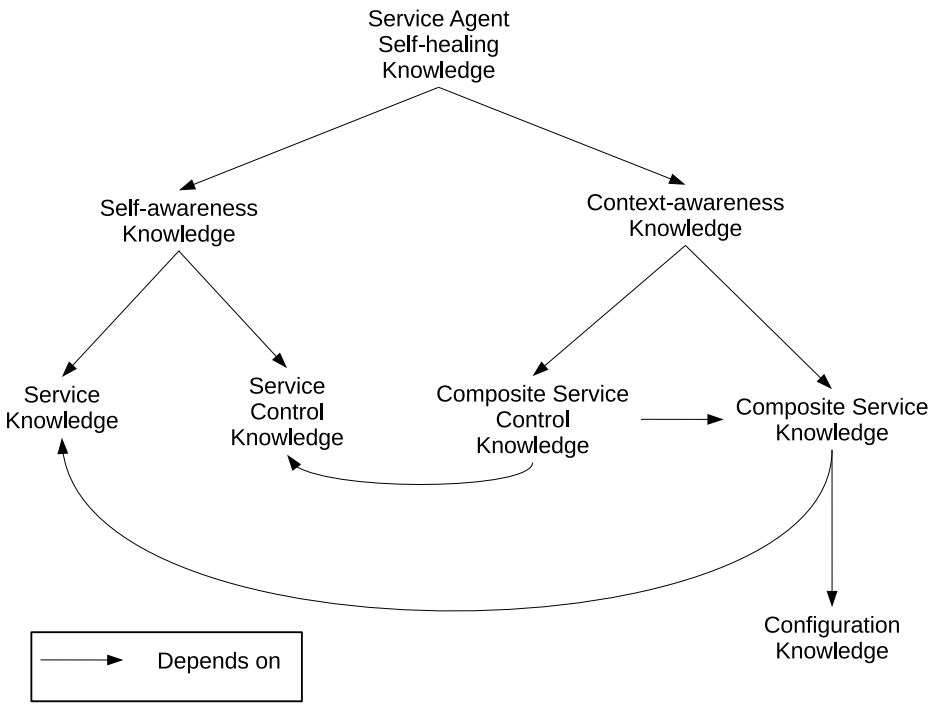


Figure 4.6: Service Agent Knowledge.

Agent. Figure 4.6 shows the Service Agent knowledge classification and dependency. The Service Agent knowledge about its self-haling state depends on its self- and context knowledge. The self-awareness knowledge is the information Service Agents have about their corresponding elementary service and their execution control. The context-awareness knowledge is composed by the information about the CPN-EP (Def. 3.1.1) computed by the Agent Coordinator, and configuration information. Finally, the composite service control knowledge uses information from the CPN-EP knowledge and from the service control knowledge.

The Agent Coordinator may always compute the Composite Service Knowledge since it knows the composite service Colored Petri net structure and the functional and nonfunctional properties of services, which is the Service Knowledge. Then, the Agent Coordinator may traverse the composite service Colored Petri net using Breadth-first search or Depth-first search-based algorithms. Finally, the Composite Service Control Knowledge and Service Control Knowledge may be computed by Service Agents at runtime by monitoring their execution.

Note that, in contrast with Chapter 3, in this chapter we model composite services using the Service Agent notation (γ) instead of services (s) since we focus on Service Agents. Also, $\gamma_\Diamond$ and $\gamma_\blacklozenge$ are never taken into account for the

computations presented in the next sections. They are only used as control Service Agents to define the beginning and the end of the composite service. This is the same as if they had ideal QoS values: 0 execution time and price, and 1 for availability.

4.3.1 Self-awareness Knowledge

The self-awareness definitions correspond to the knowledge Service Agents have on their own, without knowing anything about other Service Agents participating in a composite service execution. We classify the self-awareness knowledge into two categories: *Service Knowledge*, and *Control Knowledge*.

4.3.1.1 Service Knowledge

Independently of the technique used for QoS criteria estimation, we assume that each service is annotated with the QoS criteria introduced in Section 2.1: estimated execution time, price, and availability. Services are also annotated with their transactional property. This set of properties inherent to services are the most basic ones in the category of self-knowledge.

The list of functionally equivalent services is also part of the self-knowledge. Services in this list are functionally equivalent to the current service; however, their inputs, outputs, QoS, and transactional property can vary. These services can be used to replace or replicate the current service. We adopt the following Service Agent-based notation to reflect what behavior is expected from a Service Agent:

- $\gamma^{\approx}_{time}$: estimated execution time (Section 2.1);
- $\gamma^{\approx}_{price}$: price (Section 2.1);
- $\gamma^{\approx}_{availability}$: availability (Section 2.1);
- γ_{tp}: transactional property (Section 2.2.2);
- $\gamma_{services}$: the list transactional replacement services (Def. 3.1.15).

For example, if we say "Service Agent execution time", it is the same as saying "the execution time of the corresponding service of a Service Agent ". With a slight abuse of notation, we will refer to Service Agents γ instead of services s.

4.3.1.2 Control Knowledge

Control knowledge refers to information about the monitoring of the current local execution. The first definition concerns the execution state of a Service Agent:

Definition 4.3.1 ***Service Agent Execution State*** **(*γ_{state}*).** *The execution state of a Service Agent reflects the execution state of its corresponding service. Its possible values are: I: initial; F: fireable; R: running, E: executed; C: compensated; X: failed; and G: fixed. This definition extends the execution states of Def. 3.1.7 and places this information as part of Service Agents knowledge instead of only BRCPN-EP execution control.*

The second type of control knowledge definitions is a set of QoS related definitions. These definitions reflect the consumed QoS during the current execution, and they are defined as follows:

Definition 4.3.2 ***Local Consumed Time*** **(*γ_{time}*).** *It is the runtime counterpart definition of $\gamma^{\approx}_{time}$; therefore, it is the time a Service Agent spends executing its corresponding service. γ_{time} starts running when γ receives the necessary inputs to invoke its corresponding service, and it can be measured at any moment; for example, during a service execution, or after a successful or failed service execution.*

Definition 4.3.3 ***Local Consumed Price*** **(*γ_{price}*).** *It is the runtime counterpart definition of $\gamma^{\approx}_{price}$; it is the actual charged price after a service execution.*

Definition 4.3.4 ***Local Consumed Availability*** **(*$\gamma_{availability}$*).** *It is the runtime counterpart definition of $\gamma^{\approx}_{availability}$; it is the actual availability after a service execution.*

4.3.2 Context-awareness Knowledge

We define context as the current configuration and state of a composite service execution. Thus, context-awareness is the knowledge a Service Agent has about other Service Agents participating in the same composite service execution. The

most basic context-knowledge comes from the estimations of the global QoS of the composite service. The most interesting global QoS estimation is the one regarding execution time since it depends on the CPN-EP structure; price and availability do not depend on the CPN-EP structure.

4.3.2.1 Composite Service Knowledge

The composite service knowledge refers to the information that can be computed by knowing the CPN-EP structure and its component services. This knowledge is static and it does not depend on the composite service execution, so it can be precomputed offline by the Agent Coordinator and distributed to Service Agents before execution.

The first definition refers to the estimated execution time of the whole composite service. Finding this estimated execution time is essentially computing the longest path, in terms of service execution time, of the CPN-EP. This problem has a well known application for project planning, scheduling, and coordination, and it is known as the Critical Path [49]. The critical path of a graph can be computed using a brute-force algorithm, the Bellman-Ford algorithm [21]; however, it has a linear time solution for directed acyclic graphs using topological sorting. Ideally, the CPN-EP structure will be annotated with this information at composite service time.

Definition 4.3.5 ***Global Execution Time*** **($\circ^{\approx}_{time}$).** *It is the estimated execution time of the composite service; that is, the sum of the execution times of Service Agents in the Critical Path of the CPN-EP:*

$$\circ^{\approx}_{time} = \sum_{i} \gamma^{\approx}_{i_{time}} |\ \gamma_i \in CriticalPath \tag{4.1}$$

Figure 4.7 shows an example composite service along with the QoS of its component Service Agents. Service Agents in diamonds are in the critical path of the CPN-EP. By Def. 4.3.5, we have that:

$$\begin{aligned} \circ^{\approx}_{time} &= \gamma^{\approx}_{\Diamond_{time}} + \gamma^{\approx}_{1_{time}} + \gamma^{\approx}_{3_{time}} + \gamma^{\approx}_{6_{time}} + \gamma^{\approx}_{\blacklozenge_{time}} \\ &= 30\ ms \end{aligned} \tag{4.2}$$

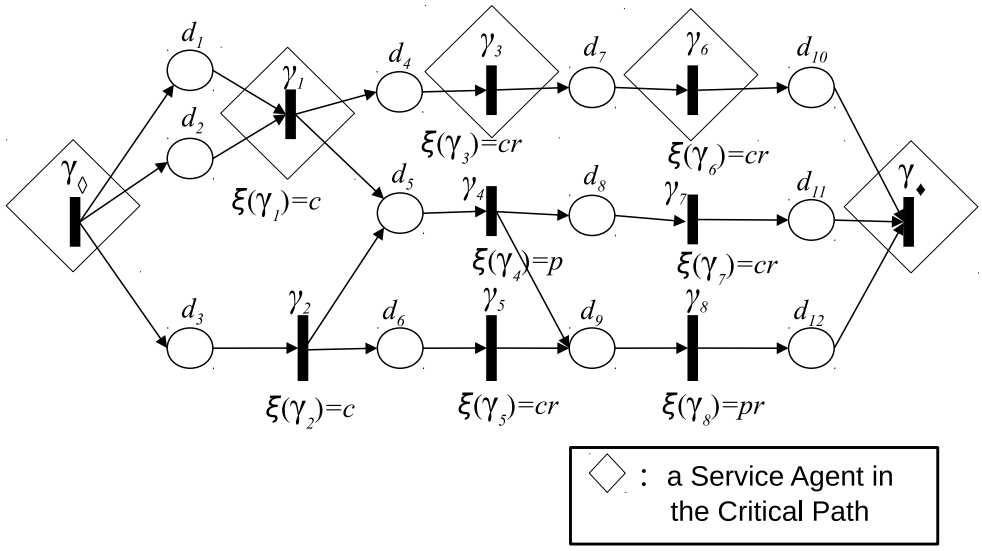


Figure 4.7: Example Composite Service with Critical Path.

	$\gamma^{\approx}_{time}$ (ms)	$\gamma^{\approx}_{price}$	$\gamma^{\approx}_{availability}$
γ_1	10	2	0.9
γ_2	7	4	0.85
γ_3	11	3	0.91
γ_4	3	2	0.82
γ_5	4	2	0.87
γ_6	9	5	0.9
γ_7	6	1	0.8
γ_8	5	4	0.88

Table 4.1: QoS of the Composite Service of Figure 4.7.

Note that, even though $\gamma_\diamond$ and $\gamma_\blacklozenge$ appear in Eq. 4.2, they are systematically excluded from actual computations. We can easily verify that no other path in the composite service has a greater time than 30. Other global QoS values, such as global price and availability, can be computed in simpler ways. The global price of a composite service is defined as follows:

Definition 4.3.6 ***Global Price*** **($\circ^{\approx}_{price}$).** *$\circ^{\approx}_{price}$ is the sum of prices of all Service Agents, defined as:*

$$\circ^{\approx}_{price} = \sum_i \gamma^{\approx}_{i_{price}} \tag{4.3}$$

Following the example, we have that $\circ^{\approx}_{price} = 23$.

The global availability is the product of all availabilities, as follows:

Definition 4.3.7 ***Global Availability*** ***($\circ^{\approx}_{availability}$).*** *It is the probability that the composite service executes without failures. It is calculated as follows:*

$$\circ^{\approx}_{availability} = \prod_{i} \gamma^{\approx}_{i_{availability}}$$

In our example, $\circ^{\approx}_{availability} = 0.314$. This means that the example composite service has 68.6 percent probability of failure.

Having these global QoS values, Service Agents may know if they may consume more QoS than previously expected. We call *available QoS* the *extra* QoS a Service Agent may consume. Once again, the computation of the available execution time for a Service Agent is particular, different from the available price or the available availability. Before presenting the definition of the available execution time, we have to introduce three other definitions: expected firing time; remaining time; and local critical path. The expected firing time is defined as:

Definition 4.3.8 ***Expected Firing Time*** ***($\gamma^{\approx}_{firing}$).*** *It is the estimated time from the beginning of the composite service execution until the corresponding service of γ is fireable. It is defined recursively as follows:*

$$\gamma^{\approx}_{firing} = \begin{cases} 0 & if \ \forall d \in {}^{\bullet}\gamma, \ d \in CPN\text{-}EP_{IN} \\ max(\gamma^{\approx}_{i_{firing}} + \gamma^{\approx}_{i_{time}}) | \gamma_i \in {}^{\bullet}({}^{\bullet}\gamma) & if \ \exists d \in {}^{\bullet}\gamma | \ d \notin CPN\text{-}EP_{IN} \end{cases}$$

Let us take the Service Agent γ_4 of Figure 4.7, on page 85, as example. Since the execution time of $\gamma_{\diamond}$ is 0, its expected firing time is given by the maximum execution time between γ_1 and γ_2; that is, $\gamma^{\approx}_{(4)firing} = 10$ ms. Similarly, the remaining execution time is defined as follows:

Definition 4.3.9 ***Remaining Execution Time*** ***($\gamma^{\approx}_{remaining}$).*** *It is the estimated execution time from the end of a Service Agent execution until the end of the composite service. It is recursively defined as follows:*

$$\gamma^{\approx}_{remaining} = \begin{cases} 0 & if \ \forall d \in \gamma^{\bullet}, d \in CPN\text{-}EP_{OUT} \\ max(\gamma^{\approx}_{i_{time}} + \gamma^{\approx}_{i_{remaining}}) | \gamma_i \in (\gamma^{\bullet})^{\bullet} & if \ \exists d \in \gamma^{\bullet} | d \notin CPN\text{-}EP_{OUT} \end{cases}$$

The remaining time from γ_4 is then the critical path from it until the end of the composite service; that is, the sum of the execution times of γ_6 and $\gamma_\blacklozenge$, so $\gamma^{\approx}_{(6)remaining} = 9$ ms. Then, we can compute the local global time:

Definition 4.3.10 ***Local Global Time*** **(*$\gamma(\circ)^{\approx}_{time}$*).** *It is the expected composite service execution time from the point of view of a Service Agent γ. $\gamma(\circ)^{\approx}_{time}$ is based on the expected firing time (Def. 4.3.8), the estimated execution time ($\gamma^{\approx}_{time}$), and the remaining execution time (Def. 4.3.9). It is calculated as follows:*

$$\gamma(\circ)^{\approx}_{time} = \gamma^{\approx}_{firing} + \gamma^{\approx}_{time} + \gamma^{\approx}_{remaining} \tag{4.4}$$

Note that, by Def. 4.3.5, the local global time $\gamma(\circ)^{\approx}_{time}$ always satisfies:

$$\gamma(\circ)^{\approx}_{time} \leq \circ^{\approx}_{time}$$

Also, by Def. 4.3.5, for services which are not in the Critical Path we have that:

$$\gamma(\circ)^{\approx}_{time} < \circ^{\approx}_{time}$$

and for services in the critical path:

$$\gamma(\circ)^{\approx}_{time} = \circ^{\approx}_{time} \tag{4.5}$$

Then, the local global time for γ_4 is:

$$\begin{aligned}\gamma_4(\circ)^{\approx}_{time} &= 10 + 3 + 9 \\ &= 22\ ms \\ &< \circ^{\approx}_{time} \\ &< 30\ ms\end{aligned}$$

Finally, by knowing the estimations of the global time $\circ^{\approx}_{time}$ and the local global time $\gamma(\circ)^{\approx}_{time}$, Services Agents can compute their available extra time they can use without affecting the composite service.

Definition 4.3.11 ***Available Time*** **(γ^{+}_{time}).** *It represents the time a Service Agent can consume additionally to the expected execution time of its corresponding service. It is computed by subtracting the local global time to the global time as follows:*

$$\gamma^{+}_{time} = \circ^{\approx}_{time} - \gamma(\circ)^{\approx}_{time} \tag{4.6}$$

The available time for γ_4 is $\gamma^{+}_{4_{time}} = 30 - 22 = 8$ ms. This means that γ_4 can use extra 8 ms; for example, taking more execution time or performing service replacement. By Equation 4.5, we have that the initial available time for Service Agents in the Critical Path is:

$$\gamma^{+}_{time} = 0 |\ \gamma \in CriticalPath$$

Given the nature the computation of other QoS such as price (Def. 4.3.6) and availability (Def. 4.3.7), the concept of QoS availability does not exist as for the execution time since QoS variation influences directly the global QoS.

Definition 4.3.12 ***Available Price*** **(γ^{+}_{price}).** *It represents the price a Service Agent can consume additionally to the expected price of its corresponding service. It is computed as follows:*

$$\gamma^{+}_{price} = \circ_{price} - \left(\sum_i \gamma_{i^{\approx}_{price}} [\gamma_{i_{price}} = 0] + \sum_i \gamma_{i_{price}} [\gamma_{i_{price}} \neq 0] \right) \tag{4.7}$$

The available availability follows similarly:

Definition 4.3.13 ***Available Availability*** **($\gamma^{+}_{availability}$).** *It represents the availability a Service Agent can consume additionally to the expected availability of its corresponding service. It is computed as follows:*

$$\gamma^{+}_{availability} = \circ_{availability} - \left(\prod_i \gamma_{i^{\approx}_{availability}} [\gamma_{i_{availability}} = 0] \times \prod_i \gamma_{i_{availability}} [\gamma_{i_{availability}} \neq 0] \right) \tag{4.8}$$

Note that, in an ideal execution where all Service Agents consume the same price and availability than expected, we have that:

$$\circ_{price} = \left(\sum_i \gamma_{i^{\approx}_{price}}[\gamma_{i_{price}} = 0] + \sum_i \gamma_{i_{price}}[\gamma_{i_{price}} \neq 0] \right)$$

and,

$$\circ_{availability} = \left(\prod_i \gamma_{i^{\approx}_{availability}}[\gamma_{i_{availability}} = 0] \times \prod_i \gamma_{i_{availability}}[\gamma_{i_{availability}} \neq 0] \right)$$

therefore,

$$\gamma^{+}_{price} = \gamma^{+}_{availability} = 0 \tag{4.9}$$

Service Agents are also aware of information not related to QoS. The first set of definitions provides a way of knowing the progress of the whole composite execution from the point of view of Service Agents. The first of these definitions regards the composite service outputs which depend on a Service Agent:

Definition 4.3.14 ***Dependent and Nondependent Output*** **(**$\gamma_{d-outputs}$**,** $\gamma^{-1}_{d-outputs}$**).** *$\gamma_{d-outputs}$ is the set of CPN-EP outputs that depend on a successful execution of the Service Agent γ, while $\gamma^{-1}_{d-outputs}$ is the set of CPN-EP outputs that do not depend on a successful execution of the Service Agent γ; that is:*

$$\forall d \in CPN\text{-}EP_{OUT},\ d \in \gamma_{d-outputs} \iff \gamma \in pred(d)$$

and

$$\forall d \in CPN\text{-}EP_{OUT},\ d \in \gamma^{-1}_{d-outputs} \iff \gamma \notin pred(d)$$

where $pred(d)$ is the set of all predecessors of d. This degree of dependent output reflects the importance of a Service Agent in terms of the number of CPN-EP outputs that depends on its successful execution. For example, in Figure 4.7, all the CPN-EP outputs depend on the successful execution of γ_4; therefore, $\gamma_{4_{d-outputs}} = \{d_{10}, d_{11}, d_{12}\}$.

Knowing its predecessors allows a Service Agent to evaluate the progress of a CPN-EP execution in terms of how many Service Agents have successfully executed their corresponding services. The number of predecessors of a Service Agent may be computed by traversing the CPN-EP, and it is defined as follows:

Definition 4.3.15 ***Predecessors*** **(**$\gamma_{predecessors}$**).** *It is the number of predecessors Service Agents of a given Service Agent; that is:*

$$\forall \gamma_i \in \Gamma,\ \gamma_i \in \gamma_{predecessors} \iff \gamma_i \in pred(\gamma)$$

In the example of Figure 4.7, $\gamma_{4_{predecessors}} = \{\gamma_1, \gamma_2\}$.

Service Agents have to know their possibility to initiate a compensation process to take smarter recovery decisions. This knowledge is important for retriable Service Agents since they cannot know if they can initiate a compensation process only by looking at their transactional property. The compensation ability of a Service Agent is defined as follows:

Definition 4.3.16 ***Compensation Ability*** **(**$\gamma_{compensable}$**).** *It represents the ability of a Service Agent to trigger a compensation process and it is denoted as* $\gamma_{compensable}$. $\gamma_{compensable} = TRUE$ *for a Service Agent* γ *if the following two conditions satisfy:*

1) $\forall \gamma_i \in pred(\gamma),\ \gamma_{i_{TP}} \in \{c, cr\}$

2) $\forall \gamma_i \in parallel(\gamma),\ \gamma_{i_{TP}} \in \{c, cr\}$

where $pred(\gamma)$ *is the set of all predecessor Service Agents of* γ*, and* $parallel(\gamma)$ *is the set of all Service Agents in parallel paths to* γ*. Service Agents in* $parallel(\gamma)$ *are not reachable from* γ*, and* γ *is not reachable from any of the Service Agents in* $parallel(\gamma)$*.*

In the example of Figure 4.7, we have that:

$$\begin{aligned}
\gamma_{1_{compensable}} &= TRUE \\
\gamma_{2_{compensable}} &= TRUE \\
\gamma_{3_{compensable}} &= FALSE \\
\gamma_{4_{compensable}} &= TRUE \\
\gamma_{5_{compensable}} &= FALSE \\
\gamma_{6_{compensable}} &= FALSE \\
\gamma_{7_{compensable}} &= FALSE \\
\gamma_{8_{compensable}} &= FALSE
\end{aligned}$$

Definition 4.3.17 ***Progress*** **($\gamma(\circ)_{progress}$).** *It refers to how many Service Agents have successfully executed their corresponding services from the point of view of a Service Agent. Service Agents know how many predecessors they have ($\gamma_{predecessors}$) and how many composite service outputs depend on their successful execution ($\gamma_{d-outputs}$); therefore, a Service Agent can compute the CPN-EP progress as follows:*

$$\gamma(\circ)_{progress} = \omega_{predecessors} * \frac{100 * |\gamma_{predecessors}|}{|\Gamma|} + \omega_{d-outputs} * \frac{100 * |\gamma^{-1}_{d-outputs}|}{|\Gamma|} \tag{4.10}$$

where $\omega_{predecessors}$ and $\omega_{d-outputs}$ are weights corresponding to $\gamma_{predecessors}$ and $\gamma_{d-outputs}$ respectively, and $\omega_{predecessors} + \omega_{d-outputs} = 1$.

In the case of γ_4, using $\omega_{predecessors} = \omega_{d-outputs} = 0.5$, we have that:

$$\begin{aligned}
\gamma_4(\circ)_{progress} &= 0.5 * \frac{100 * |\{\gamma_1, \gamma_2\}|}{8} + 0.5 * \frac{100 * |\{\emptyset\}|}{8} \\
&= 0.5 * 25 + 0.5 * 0.0 \\
&= 12.5 + 0.0 \\
&= 12.5\%
\end{aligned}$$

From the point of view of γ_4, the composite service execution progress is 12.5%; however, γ_4 cannot know the progress of parallel paths such as γ_3 and γ_5. Note that $\gamma_\diamond$ and $\gamma_\blacklozenge$ are never taken into account for $|\Gamma|$.

4.3.2.2 Composite Service Control Knowledge

So far, we have presented the definitions regarding estimated QoS and static information. Now, we are going to present their corresponding runtime definitions. The global consumed time is the firing time of a Service Agent as follows:

Definition 4.3.18 ***Firing Time*** **(*γ_{firing}*).** *It is the runtime counterpart definition of Def. 4.3.8. γ_{firing} is the time passed from the beginning of the CPN-EP execution until γ is fireable. It is the corresponding actual value of the estimation $\gamma^{\approx}_{firing}$.*

$$\gamma_{firing} = \begin{cases} 0 & if \ \forall d \in {}^{\bullet}\gamma, \ d \in CPN\text{-}EP_{IN} \\ max(\gamma_{i_{firing}} + \gamma_{i_{time}}) | \gamma_i \in {}^{\bullet}({}^{\bullet}\gamma) & if \ \exists d \in {}^{\bullet}\gamma | \ d \notin CPN\text{-}EP_{IN} \end{cases}$$

The global consumed time is defined as follows:

Definition 4.3.19 ***Global Consumed Time*** **(*$\circ_{time}$*).** *It is the runtime counterpart definition of Def. 4.3.5. It is the real time taken to execute a CPN-EP. It is the firing time (Def. 4.3.18) of $\gamma_{\blacklozenge}$; thus, it is measured only when all Service Agents have finished as follows:*

$$\circ_{time} = \gamma_{\blacklozenge_{firing}}$$

The global consumed price is defined as:

Definition 4.3.20 ***Global Consumed Price*** **(*$\circ_{price}$*).** *It is the runtime counterpart definition of Def. 4.3.6. $\circ_{price}$ is the actual price of the CPN-EP at any moment of its execution. At the end of the CPN-EP execution, it is the corresponding actual value of the estimation $\circ^{\approx}_{price}$.*

$$\circ_{price} = \sum_i \gamma_{i_{price}}$$

and the global consumed availability is:

Definition 4.3.21 ***Global Consumed Availability*** ***($\circ_{availability}$).*** *It is the runtime counterpart definition of Def. 4.3.7. It is the actual availability of the CPN-EP at any moment of its execution. At the end of the CPN-EP execution, it is the corresponding actual value of the estimation $\circ^{\approx}_{availability}$.*

$$\circ_{availability} = \prod_i \gamma_{i_{availability}}$$

Service Agents can then recompute their available QoS using the actual consumed one. The following definitions represent the updated available QoS values:

Definition 4.3.22 ***Updated Local Global Time*** ***($\gamma(\circ)_{time}$).*** *It is the runtime counterpart definition of Def. 4.3.10. It is calculated as follows:*

$$\gamma(\circ)_{time} = \gamma_{firing} + \gamma^{\approx}_{time} + \gamma^{\approx}_{remaining} \tag{4.11}$$

Note that Equation 4.3.2.2 represents the updated local global time when a Service Agent is fireable; however, it can change depending of the Service Agent execution state. For example, after the Service Agent executed successfully:

$$\gamma(\circ)_{time} = \gamma_{firing} + \gamma_{time} + \gamma^{\approx}_{remaining}$$

where γ_{time} is the time taken by the successful execution. If its corresponding service fails, then the equation will be expressed as:

$$\gamma(\circ)_{time} = \gamma_{firing} + \gamma_{time} + \gamma^{\approx}_{time} + \gamma^{\approx}_{remaining}$$

where γ_{time} is the time of the first failed execution.

In the same way, the available time (Equation 4.6) of a Service Agent can also be updated using $\gamma(\circ)_{time}$ instead of $\gamma(\circ)^{\approx}_{time}$ as follows:

$$\gamma^{+}_{time} = \circ^{\approx}_{time} - \gamma(\circ)_{time}$$

4.3.2.3 Configuration Knowledge

Configuration knowledge comprises the system parameters defining acceptable system states and it may be set by users. We introduce thresholds for the global execution time, global price, local availability, and global progress. We consider that the values of these thresholds are given as user preferences. They are defined as follows:

Definition 4.3.23 ***Time Threshold*** **(*θ_{time}*).** *It is the threshold for the CPN-EP execution time $\circ_{time}$ (Def. 4.3.19); that is, it is the maximum allowed execution time for a CPN-EP execution, where:*

$$\theta_{time} \in [\circ^{\approx}_{time}, +\infty]$$

with $+\infty$ as default value.

Definition 4.3.24 ***Price Threshold*** **(*θ_{price}*).** *It is the threshold for the CPN-EP price $\circ_{price}$ (Def. 4.3.20); that is, it is the maximum allowed price for a CPN-EP execution, where:*

$$\theta_{price} \in [\circ^{\approx}_{price}, +\infty]$$

with 0 as default value.

Definition 4.3.25 ***Availability Threshold*** **(*$\theta_{availability}$*).** *It is the threshold for the CPN-EP availability $\circ_{availability}$ (Def. 4.3.21); that is, it is the maximum allowed availability for a CPN-EP execution, where:*

$$\theta_{availability} \in [0, \circ^{\approx}_{availability}] \tag{4.12}$$

with 0 as default value.

We have shown in Eq. 4.3.2.1 and Eq. 4.9 that, initially,

$$\gamma^{+}_{time} = \gamma^{+}_{price} = \gamma^{+}_{availability} = 0$$

Using the thresholds θ_{time}, θ_{price} and $\theta_{availability}$, we may let γ^{+}_{time}, γ^{+}_{price}, and $\gamma^{+}_{availability}$ become positive and, at the same time, we set a maximum value for their corresponding QoS criteria. For example, a Service Agent may replace its corresponding service with a service with higher price or lower availability, but staying within the acceptable values imposed by thresholds θ_{price} and $\theta_{availability}$.

Definition 4.3.26 ***Replication Threshold*** **(*$\theta_{replication}$*).** *It is a threshold for replicating according to $\gamma_{availability}$. The service availability threshold may be used by a Service Agent to trigger replication if the availability of its corresponding service is lower than the threshold value. It is defined as:*

$$\theta_{replication} \in [-\infty, \gamma^{\approx}_{availability}]$$

with $-\infty$ as default value.

Definition 4.3.27 ***Progress Threshold*** **(*$\theta_{progress}$*).** *It is the threshold for the CPN-EP progress $\gamma(\circ)_{progress}$ (Def. 4.3.17), where:*

$$\theta_{progress} \in [0, 100]$$

with ∞ as default value.

4.3.2.4 Self-healing State Knowledge

The self-healing state knowledge is the state of the system expressed as one of the three self-healing states: normal, degraded, or broken. The state of the system is given by the values of the previously presented definitions. The self-healing state of a Service Agent is defined as follows:

Definition 4.3.28 ***Service Agent Self-healing State*** **(*$\gamma_{sh-state}$*).** *It refers to the self-healing state of a Service Agent, where:*

$$\gamma_{sh-state} \in \{NORMAL, DEGRADED, BROKEN\}$$

Definition 4.3.29 ***Composite Service Self-healing State*** ***($\circ_{sh-state}$).*** *It refers to the self-healing state of a composite service which depends on the self-healing states of component Service Agents:*

$$\circ_{sh-state} = \begin{cases} NORMAL & if\ \forall\gamma \in \Gamma,\ \gamma_{sh-state} = NORMAL \\ DEGRADED & if\ \exists\gamma \in \Gamma |\ \gamma_{sh-state} = DEGRADED \\ BROKEN & if\ \exists\gamma \in \Gamma |\ \gamma_{sh-state} = BROKEN \end{cases}$$

Note that Def. 4.3.29 does not specify how to find the self-healing state, it only defines the possible values of $\gamma_{sh-state}$. In the next section, Section 4.4, we explain how to deduce the value of $\gamma_{sh-state}$ given the self- and context-knowledge information.

4.3.2.5 Summary of Service Agent Knowledge

To summarize, Table 4.2 shows the knowledge of Service Agents classified by self- or context-awareness. Regarding the notation, the most important aspects to remember are that γ represents a Service Agent, $\circ$ represents the composite service, estimated values are marked with $^{\approx}$, and that $\gamma(\circ)$ represents the global composite service view of a Service Agent. Also, remember our abuse of notation: we talk mainly about Service Agents instead of services.

4.4 Knowledge Base

Service Agents are equipped with a knowledge base containing the information presented in Section 4.3.1 and Section 4.3.2. The knowledge base also contains a set of rules to deduce the QoS state, the self-healing state, and the recovery actions for a composite service execution. The Service Agent knowledge base is defined as follows:

Definition 4.4.1 ***Service Agent Knowledge Base*** ***($\mathcal{KB}_\gamma$).*** *The Service Agent knowledge base is a triplet*

$$\mathcal{KB}_\gamma = (\Delta_\gamma, \phi_\gamma, \mathcal{R}_\gamma) \tag{4.13}$$

	Awareness		Definition
	Self-	*Context-*	
$\gamma^{\approx}_{time}$,$\gamma^{\approx}_{price}$, $\gamma^{\approx}_{availability}$	✓		Section 2.1
γ_{tp}	✓		Section 2.2.2
$\gamma^{\approx}_{services}$	✓		Def. 3.1.15
γ_{state}	✓		Def. 4.3.1
I	✓		Initial state of a Service Agent Def. 4.3.1
F	✓		Fireable state of a Service Agent Def. 4.3.1
R	✓		Running state of a Service Agent Def. 4.3.1
E	✓		Executed state of a Service Agent Def. 4.3.1
C	✓		Compensated state of a Service Agent Def. 4.3.1
X	✓		Failed state of a Service Agent Def. 4.3.1
G	✓		Fixed state of a Service Agent Def. 4.3.1
γ_{time},γ_{price},$\gamma_{availability}$	✓		Def. 4.3.2, 4.3.3, 4.3.4
$\circ^{\approx}_{time}$,$\circ^{\approx}_{price}$,$\circ^{\approx}_{availability}$		✓	Def. 4.3.5, 4.3.6, 4.3.7
$\gamma^{\approx}_{firing}$		✓	Def. 4.3.8
$\gamma^{\approx}_{remaining}$		✓	Def. 4.3.9
$\gamma(\circ)^{\approx}_{time}$		✓	Def. 4.3.10
γ^{+}_{time},γ^{+}_{price}, $\gamma^{+}_{availability}$		✓	Def. 4.3.11, 4.3.12, 4.3.13
γ_{firing}		✓	Def. 4.3.18
$\circ_{time}$, $\circ_{price}$, $\circ_{availability}$		✓	Def. 4.3.19, 4.3.20, 4.3.21
$\gamma(\circ)_{time}$		✓	Def. 4.3.22
$\gamma_{d-outputs}$, $\gamma^{-1}_{d-outputs}$		✓	Def. 4.3.14
$\gamma_{predecessors}$		✓	Def. 4.3.15
$\gamma_{compensable}$		✓	Def. 4.3.16
$\gamma(\circ)_{progress}$		✓	Def. 4.3.17
θ_{time}, θ_{price}, $\theta_{availability}$		✓	Def. 4.3.23, 4.3.24, 4.3.25
$\theta_{replication}$		✓	Def. 4.3.26
$\theta_{progress}$		✓	Def. 4.3.27
$\gamma_{sh-state}$		✓	Def. 4.3.29

Table 4.2: Summary of Service Agent Knowledge Classification.

where Δ_γ is the set of possible conclusions, ϕ_γ is a set of observable findings, and $\mathcal{R}_\gamma$ is a set of rules.

The set of observable findings ϕ_γ corresponds to the Service Agent knowledge we have described in this chapter.

The conclusions Δ_γ a Service Agent can deduce are related to preventive and corrective actions. A Service Agent action may also be to continue its normal execution. The Service Agent possible conclusions are the following:

$$\Delta_\gamma = \{CONTINUE,\ RETRY,\\ REPLACE,\ COMPENSATE,\\ CHECKPOINT,\ REPLICATE\}$$

where $CONTINUE$ means not taking any action and continue the execution as normal; $RETRY$ refers to the service retrying mechanism for *retriable* services explained in Section 3.2.1.2; $REPLACE$ is the service replacement mechanism presented in Section 3.1.3; $COMPENSATE$ the compensation mechanism explained in Section 3.1.1; $CHECKPOINT$ the checkpointing mechanism of Section 3.1.2; and $REPLICATE$ is an active replication mechanism where all transactional equivalent services are invoked in parallel and the first returned response is taken as the final result. For a set of transactional equivalent services $s_\equiv$, if we invoke them using active replication, the expected execution time is the minimum expected execution time among the replicas:

$$\gamma^{\approx}_{time} = min(s^{\approx}_{i_{time}}) \mid s_i \in s_\equiv$$

and the expected availability is computed as follows:

$$\gamma^{\approx}_{availability} = 1 - \prod_{i}^{|s_\equiv|}(1 - s^{\approx}_{i_{availability}}) \mid s_i \in s_\equiv$$

$\mathcal{R}_\gamma$ is a set of inference rules to allow Service Agents deducing actions. Premises belong to ϕ_γ and conclusions to Δ_γ. These rules follow the standard form:

$$\frac{\begin{array}{c} premise_1 \in \phi_\gamma \\ premise_2 \in \phi_\gamma \\ \dots \\ premise_n \in \phi_\gamma \end{array}}{conclusion \in \Delta_\gamma}$$

The ultimate goal of the knowledge base is to deduce an action given a system state. The finding of a Service Agent action is based on its self-healing state. Before presenting the rules to transition between self-healing states, we present two QoS related rules: one rule to deduce if the composite service execution QoS

is worse than expected; another rule to know if the composite service execution QoS is not acceptable. Then, we present the rules to deduce self-healing states, and the rules to deduce actions.

Note that we do not intend these rules to satisfy the needs of every user and system. Instead, we present them to illustrate the usage of the Service Agent knowledge we define in this chapter. A real-world implementation of our system should provide rule customization, verification, and analysis.

4.4.1 QoS State Deduction

Service Agents can have an idea of the QoS state of the composite service. They can know if the QoS state is a degraded but an acceptable one; or an unacceptable QoS state. Service Agents default QoS state is $Normal_{QoS}$, which means that there is no QoS criterion worse than expected:

$$\mathcal{R}_{QoS^1} : \frac{\nexists QoS_i \in QoS | \circ_{QoS_i} > \circ^{\approx}_{QoS_i}}{Normal_{QoS}}$$

During execution, the $Normal_{QoS}$ state may change. We say that the QoS is degraded if for at least one QoS criterion $QoS_i \in \{time, price, availability\}$, its actual global value $\circ_{qos}$ is greater than its expected value $\circ^{\approx}_{QoS_i}$, as follows:

$$\mathcal{R}_{QoS^2} : \frac{\begin{array}{c}\exists QoS_i \in QoS | \circ_{QoS_i} > \circ^{\approx}_{QoS_i} \\ \nexists QoS_i \in QoS | \circ_{QoS_i} > \theta_{QoS_i}\end{array}}{Degraded_{QoS}}$$

We say that the QoS is broken if for at least one QoS criterion QoS_i, its actual global value $\circ_{qos}$ is greater than its threshold value θ_{QoS_i} as follows:

$$\mathcal{R}_{QoS^3} : \frac{\exists QoS_i \in QoS | \circ_{QoS_i} > \theta_{QoS_i}}{Broken_{QoS}}$$

$\mathcal{R}_{QoS^1}$ compares the expected QoS values(Def 4.3.5, Def 4.3.6, and Def 4.3.7) with their counterpart runtime values (Def 4.3.22, Def 4.3.21, and Def 4.3.20). $\mathcal{R}_{QoS^2}$ compares the QoS runtime values with their corresponding thresholds.

4.4.2 Self-healing State Deduction

The self-healing state of a Service Agents depends on the QoS state (Section 4.4.1) and the execution state γ_{state} (Def. 4.3.1). In the following, we present a set of rules to deduce the self-healing state of a Service Agent, which is derived from the definition given in Section 4.2.

If the execution cannot continue without violating the threshold value for a QoS criterion, then Service Agents go to a *broken* state:

$$\mathcal{R}_{state^1} : \frac{\begin{array}{c} Broken_{QoS} \\ \gamma_{state} \notin \{C, I, A\} \end{array}}{BROKEN}$$

Service Agents go to a *degraded* state if the QoS state is $Normal_{QoS}$ but a service has failed:

$$\mathcal{R}_{state^2} : \frac{\begin{array}{c} Normal_{QoS} \\ \gamma_{state} = X \end{array}}{DEGRADED}$$

It may also go to a *degraded* state if the QoS remains under the acceptable values (Def. 4.3.23, Def. 4.3.24, and Def. 4.3.25) but at least one of the QoS criteria is worse than estimated:

$$\mathcal{R}_{state^3} : \frac{\begin{array}{c} \gamma_{state} \in \{X, G, F, R, E\} \\ Degraded_{QoS} \end{array}}{DEGRADED}$$

We consider that a Service Agent may go back to a *normal* state after compensation:

$$\mathcal{R}_{state^4} : \frac{\gamma_{state} \in \{C, I, A\}}{NORMAL}$$

or after fixing its corresponding failed service, or running without failures, if the QoS is not degraded:

$$\mathcal{R}_{state^5} : \frac{\begin{array}{c} \gamma_{state} \in \{G, F, R, E\} \\ Normal_{QoS} \end{array}}{NORMAL}$$

4.4.3 Action Deduction

We classified the action deduction rules according to the self-healing state; that is, actions from the *broken*, *normal*, and *degraded* states.

Before presenting each rule, we give a brief description of the circumstances in which a Service Agent should make a decision. The set of these circumstances represents the use cases we identify from our transactional approach of Chapter 3, and from our proposed QoS-aware approach using knowledge-based Service Agents.

Action Deduction from the Broken State

We have seen that a Service Agent is in the *broken* state if the execution cannot continue without violating the threshold value for a QoS criterion (rule $\mathcal{R}_{state^1}$). A Service Agent may trigger a compensation process if:

$$\mathcal{R}_{actions^1} : \frac{\begin{array}{c}\gamma_{sh-state} = BROKEN \\ \gamma_{compensable} \\ \gamma(\circ)_{progress} < \theta_{progress} \vee \neg checkpointingEnabled\end{array}}{COMPENSATE}$$

We have proposed the checkpointing mechanism as an alternative to compensation; therefore, it is only taken into account if a Service Agent may trigger a compensation process but the composite service execution has advanced *too much to compensate* ($\gamma(\circ)_{progress} \geq \theta_{progress}$):

$$\mathcal{R}_{actions^2} : \frac{\begin{array}{c}\gamma_{sh-state} = BROKEN \\ \gamma_{state} = X \\ \gamma_{compensable} \\ \gamma(\circ)_{progress} \geq \theta_{progress} \\ checkpointingEnabled\end{array}}{CHECKPOINT}$$

The rest of the rules concern a Service Agent which cannot trigger a compensation process; that is, the only option is continue the execution by replicating, or retrying. A Service Agent chooses replication if there are available replicas and there is no available time:

$$\mathcal{R}_{actions^3} : \frac{\begin{array}{c}\gamma_{sh-state} = BROKEN \\ \gamma_{state} \in \{F, X\} \\ \neg\gamma_{compensable} \\ s_{\equiv} \neq \emptyset \\ \gamma^{+}_{time} \leq 0\end{array}}{\text{REPLICATE}}$$

A Service Agent chooses to retry if its corresponding service failed, there are no replicas or there is available time:

$$\mathcal{R}_{actions^4} : \frac{\begin{array}{c}\gamma_{sh-state} = BROKEN \\ \gamma_{state} = X \\ \neg\gamma_{compensable} \\ \gamma^{+}_{time} > 0 \ \vee \ s_{\equiv} = \emptyset\end{array}}{RETRY}$$

For other execution states, the action is continue:

$$\mathcal{R}_{actions^5} : \frac{\begin{array}{c}\gamma_{sh-state} = BROKEN \\ \gamma_{state} \notin \{F, X\} \\ \neg\gamma_{compensable}\end{array}}{CONTINUE}$$

Action Deduction from the Degraded State

From the *degraded* state (rules $\mathcal{R}_{state^2}$ and $\mathcal{R}_{state^3}$), the only case when a Service Agent chooses compensation is when its corresponding service failed and it cannot do forward recovery:

$$\mathcal{R}_{actions^6} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = X \\ \gamma_{tp} \notin \{pr, ar, cr\} \\ s_{\equiv} = \emptyset\end{array}}{COMPENSATE}$$

Note that $\gamma_{tp} \notin \{pr, ar, cr\} \rightarrow \gamma_{compensable} = TRUE$.

If its corresponding service failed, there is available time, and the availability of its corresponding service does not require replication, it chooses retry or replace according to its transactional property and the availability of transactional equivalent services:

$$\mathcal{R}_{actions^7} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = X \\ \gamma^{+}_{time} > 0 \\ \gamma_{availability} \geq \theta_{replication} \\ \gamma_{tp} \in \{pr, ar, cr\}\end{array}}{RETRY}$$

$$\mathcal{R}_{actions^8} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = X \\ \gamma^{+}_{time} > 0 \\ \gamma_{availability} \geq \theta_{replication} \\ \gamma_{tp} \notin \{pr, ar, cr\} \\ s_{\equiv} \neq \emptyset\end{array}}{REPLACE}$$

A Service Agent in the *degraded* state continues as normal if its corresponding services is fireable, there is available time, and the availability of its corresponding service does not require replication:

$$\mathcal{R}_{actions^9} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = F \\ \gamma^{+}_{time} > 0 \\ \gamma_{availability} \geq \theta_{replication}\end{array}}{CONTINUE}$$

Regardless if its corresponding service failed or has not been executed, a Service Agent in the *degraded* state chooses replication if there is no available time or the availability of its corresponding service requires replication:

$$\mathcal{R}_{actions^{10}} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} \in \{X, F\} \\ \gamma^{+}_{time} \leq 0 \ \vee \ \gamma_{availability} < \theta_{replication} \\ s_{\equiv} \neq \emptyset\end{array}}{REPLICATE}$$

However, if the corresponding service is fireable, there is no available time or the availability of its corresponding service requires replication, but there are no services to replicate, the action is continue:

$$\mathcal{R}_{actions^{11}} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = F \\ \gamma^{+}_{time} \leq 0 \ \vee \ \gamma_{availability} < \theta_{replication} \\ s_{\equiv} = \emptyset\end{array}}{CONTINUE}$$

A similar case applies if the service failed:

$$\mathcal{R}_{actions^{12}} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} = X \\ \gamma^{+}_{time} \leq 0 \ \vee \ \gamma_{availability} < \theta_{replication} \\ \gamma_{tp} \in \{pr, ar, cr\} \\ s_{\equiv} = \emptyset\end{array}}{RETRY}$$

For any other execution state γ_{state} not requiring service execution the action is continue:

$$\mathcal{R}_{actions^{13}} : \frac{\begin{array}{c}\gamma_{sh-state} = DEGRADED \\ \gamma_{state} \notin \{X, F\}\end{array}}{CONTINUE}$$

Action Deduction from the Normal State

When a Service Agent is in the *normal* state (rules $\mathcal{R}_{state^4}$ and $\mathcal{R}_{state^5}$), there are only two possible actions: continue the execution as normal, and replicate as a proactive action. A Service Agent does nothing if there is available time and the availability of its corresponding service does not require replication:

$$\mathcal{R}_{actions^{14}} : \frac{\begin{array}{c}\gamma_{sh-state} = NORMAL \\ \gamma_{state} = F \\ \gamma^{+}_{time} > 0 \\ \gamma_{availability} \geq \theta_{replication}\end{array}}{CONTINUE}$$

A Service Agent chooses replication if there is no available time or the availability of its corresponding service does requires replication:

$$\mathcal{R}_{actions^{15}} : \frac{\begin{array}{c}\gamma_{sh-state} = NORMAL \\ \gamma_{state} = F \\ \gamma^{+}_{time} \leq 0 \ \vee \ \gamma_{availability} < \theta_{replication}\end{array}}{REPLICATE}$$

For any other execution state γ_{state} the action is continue:

$$\mathcal{R}_{actions^{16}} : \frac{\begin{array}{c}\gamma_{sh-state} = NORMAL \\ \gamma_{state} \neq F\end{array}}{CONTINUE}$$

As we said, these are rules intended to show the usage of the Service Agent knowledge and they should be customizable in a real-world implementation; however, note that rules must always comply with the composite service transactional properties. For example, a rule which conclusion is $COMPENSATE$ can never have $\neg\gamma_{compensable}$ within its premises.

4.5 QoS Manager for Summation/Product QoS Criteria

Given the nature of composite service graphs, the QoS consumption regarding execution time is different from other QoS criteria (Def. 4.3.5). A single Service Agent may take execution time based decisions regardless of what is happening elsewhere in the composite service execution. The execution time consumed by a Service Agent depends only on its predecessors and it only affects its successors (Def. 4.3.10 and Def. 4.3.22); parallel Service Agents are not affected. Summation and product based QoS, such as price (Def. 4.3.6) and availability (Def. 4.3.7), take into account all Service Agents regardless of the composite service graph structure; thus; any variation on the expected summation or product QoS criteria affects the QoS of the whole composite service. For example, we have shown in Def. 4.3.12 and Def. 4.3.13 that Service Agents must know the actual consumed QoS of all Service Agents to compute their available price (γ^{+}_{price}) and availability ($\gamma^{+}_{availability}$).

To manage summation and product based QoS consumption at runtime, the Agent Coordinator has a module in charge of managing the global QoS (Figure 4.8). In case a Service Agent needs to consume more product or summation

based QoS than expected, it contacts the Agent Coordinator to ask for QoS consumption permission. A Service Agent needs to ask for QoS consumption permission according to the following definition:

Definition 4.5.1 ***Service Agent QoS Excess.*** *Given a Service Agent γ with: QoS^{OLD}, its originally assigned QoS; $QoS^{\approx}$, its new QoS requirement. The Service Agent γ must ask for QoS consumption permission if:*

$$\exists QoS_i \in QoS^{\approx} \mid QoS_i \neq \gamma_{time} \wedge value_i > value_i^{OLD}$$

where $value_i$ is a new QoS value for a QoS criterion and $value_i^{OLD}$ its corresponding original value.

If Def. 4.5.1 satisfies, the Service Agent builds and sends a QoS consumption request to the Agent Coordinator as follows:

Definition 4.5.2 ***QoS Consumption Request*** *($QoS_{REQUEST}$)**.*** *It contains a list of QoS criteria and their requested consumption values. A QoS consumption request is a function $QoS_{REQUEST}$, such that:*

$$QoS_{REQUEST} : QoS_i \rightarrow value_i$$

where QoS_i is a summation or product QoS criterion, and $value_i$ its requested consumption value.

The decision of granting or denying a consumption request for a QoS criterion depends on its corresponding threshold, global estimation, and requested consumption value. The QoS consumption granting rules are defined as follows:

Definition 4.5.3 ***QoS Consumption Granting Rules.*** *Given a summation or product QoS criteria QoS_i, its corresponding requested consumption value $value_i$, and a requester Service Agent γ, the QoS Manager recomputes $\gamma^{+}_{QoS_i}$ by substituting the original expected value of γ for QoS_i by the requested value $value_i$. Then, using the threshold θ_{QoS_i}, the following rules apply:*

$$\theta_{QoS_i} - |\gamma^{+}_{QoS_i}| > 0 \rightarrow granted$$
$$\theta_{QoS_i} - |\gamma^{+}_{QoS_i}| \leq 0 \rightarrow denied$$

This reasoning applies for all summation and product QoS criteria such as price and availability.

Using the QoS consumption granting rules of Def. 4.5.3, the QoS manager builds a QoS consumption response which is sent by the Agent Coordinator to the requester Service Agent. The QoS consumption response is defined as follows:

Definition 4.5.4 ***QoS Consumption Response*** *($QoS_{REQUEST}$). A QoS consumption response indicates granted or denied for each QoS criterion in the request message. A consumption response is a function $QoS_{RESPONSE}$, such that:*

$$QoS_{RESPONSE} : QoS_i \rightarrow \{granted, denied\}$$

where QoS_i is a summation or product QoS criterion and $\{granted, denied\}$ the set of possible responses for a QoS consumption request.

Finally, we consider the case when a Service Agent consumes less QoS than expected. If this happens, the Service Agent sends a QoS consumption notification to the Agent Coordinator indicating a list of QoS criteria and their corresponding consumed values. A QoS consumption notification is defined as follows:

Definition 4.5.5 ***QoS Consumption Response*** *($QoS_{NOTIFICATION}$). A QoS consumption notification is a function $QoS_{NOTIFICATION}$, such that:*

$$QoS_{NOTIFICATION} : QoS_i \rightarrow value_i$$

where QoS_i is a summation or product QoS criterion and $value_i$ its corresponding consumed value.

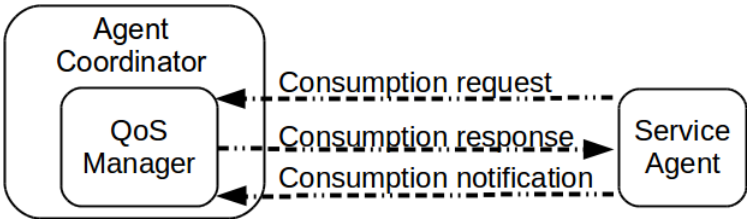


Figure 4.8: QoS Management Protocol.

4.6 Algorithms

Algorithm 4.1 shows the steps for QoS management. We present the QoS consumption request (line 1), QoS consumption notification (line 3), and Service Agent QoS consumption response (line 5) in the form of message handlers; that is, they are called when a message of their corresponding input type is received. The Service Agent QoS consumption request (line 4) is the only one which is not a message handler. It is called before invoking a service with worse QoS than expected; for example, when replacing a service for another one with higher price or lower availability.

Line 1 of Algorithm 4.1 shows the QoS consumption request message handler. For each QoS criterion in the consumption request message $QoS_{REQUEST}$ (Def. 4.5.2), it computes its available value $\gamma^{+}_{QoS_i}$ and verifies if it satisfies its corresponding threshold value θ_{QoS_i} using the QoS consumption granting rules (Def. 4.5.3). It builds the QoS consumption response indicating *granted* or *denied* for each requested QoS criterion (Def. 4.5.4). Finally, if the QoS request was granted, the QoS Manager updates the global QoS in line 2. Finally, it sends the consumption response to the requester Service Agent. Line 3 shows the QoS manager algorithm when it receives a QoS consumption notification message and it updates the global QoS.

Algorithm 4.1 line 4 shows the Service Agent consumption request algorithm. Here, the Service Agent needs the consumption of a new QoS, $QoS^{\approx}$, which is greater than its originally estimated QoS. For example, we may encounter this situation when replacing a service for another one which has higher price or lower availability. Each QoS criterion in $QoS^{\approx}$ greater than its originally estimated value in QoS^{OLD} is put in the request message $QoS_{REQUEST}$ (Def. 4.5.2); then, the consumption request is sent to the Agent Coordinator. Algorithm 4.1, line 5 shows when a Service Agent waits for a QoS consumption response $QoS_{RESPONSE}$

(Def. 4.5.4). If the request for one of the QoS criteria was denied, it returns false.

Algorithm 4.2 shows the Service Agent using of its Knowledge Base to make decisions. The initial phase and final phase are not modified from the Service Agent algorithm presented in the previous chapter (Algorithm 3.2), in Section 3.2.1. During these two phases, the Service Agent is waiting for messages, so decisions are made by other Service Agents. In line 1, the Service Agent inserts its initial knowledge in the Knowledge Base. This initial knowledge corresponds to the information that may be computed before the start of the composite service execution presented in Sections 4.3.1 and 4.3.2.

The loop of line 2 is the main control component of the Service Agent algorithm:

1) in line 3, the knowledge base is updated with new information. This information was also presented in Sections 4.3.1 and 4.3.2; however, it corresponds to what is happing at runtime. At first, the knowledge base is updated with the execution state when a Service Agent receives all the inputs it was waiting for. Then, the knowledge base may be updated as many times as needed; for example, if a service fails and a new action needs to be deduced;

2) in line 4, an action is deduced from the Knowledge Base. This action is deduced from the self- and context-knowledge using the rules presented in Section 4.4;

3) in line 5, the deduced action is executed using the algorithms presented in the previous chapter, in Section 3.2.1, for service retry, service replacement, checkpointing, and compensation. Active replication is performed by invoking all replicas and taking the result of the first one the finishes successfully.

The loop goes on until an action which makes $success = true$ is taken. For example, $success = true$ after a successful execution, compensation, triggering the checkpointing mechanism. $success$ remains $false$ when the execution of the deduced action failed; for example; the action was $CONTINUE$ or $RETRY$ but the service execution was not successful.

Algorithm 4.1 QoS Management.

```
1  Consumption Request Message Handler:
   Input: QoS_REQUEST: the consumption request message
   Input: γ: the requester Service Agent
   Output: ∅
   begin
       QoS_RESPONSE ← ∅;
       for QoS_i, value_i ∈ QoS_REQUEST do
           γ^+_{QoS_i} ← compute γ^+_{QoS_i}(value_i) //for example, using Def. 4.3.12 or 4.3.13
           //granting rules of Def. 4.5.3
           if θ_{QoS_i} − |γ^+_{QoS_i}| > 0 then
               QoS_RESPONSE[QoS_i] ← denied;
           else
               QoS_RESPONSE[QoS_i] ← granted;
           end
       end
       //update global QoS with new granted values
       if ¬QoS_RESPONSE.containsValue(denied) then
2          updateQoS(QoS_REQUEST); //Def. 4.3.20 and Def. 4.3.21
       end
       send QoS_RESPONSE to requester Service Agent;
   end
3  Consumption Notification Message Handler:
   Input: consumption_notification: the consumption notification message
   Output: ∅
   begin
       updateQoS(consumption_notification);
   end
4  Service Agent Consumption Request:
   Input: QoS^≈: the requested QoS
   Output: ∅
   begin
       QoS_REQUEST ← ∅;
       for QoS_i, value_i ∈ QoS^≈ do
           if value_i > value_i^OLD then
               QoS_REQUEST[QoS_i] ← value_i;
           end
       end
       if QoS_REQUEST ≠ ∅ then
           send QoS_REQUEST to Agent Coordinator;
       end
   end
5  Service Agent Consumption Response Message Handler:
   Input: QoS_RESPONSE: the consumption response message
   Output: ∅
   begin
       wait QoS_RESPONSE message;
       return QoS_RESPONSE.containsValue(denied);
   end
```

Algorithm 4.2 Service Agent Self-healing Code Snippet.

```
   Input: ∅
   Output: ∅
   begin
       //Initialize the Service Agent KB with self- and context-knowledge (Sections 4.3.1 and 4.3.2)
1      updateKB();
       //The Service Agent waits for its corresponding inputs (Algorithm 3.2 line 2)
       //When its corresponding service is fireable, the Service Agent
       //updates the KB and deduces an action using the following loop
       success ← false;
2      repeat
           //Insert in the KB updated self- and context-knowledge (Sections 4.3.1 and 4.3.2)
3          updateKB();
           //Deduce an action using the KB rules (Section 4.4)
4          action ← getActionKB();
           //Execute the deduced action
5          success ← Execute action;
       until success;
       Execute Final phase (Algorithm 3.3)
   end
```

4.7 Conclusions

In this chapter, we have proposed an extension to the fault tolerance model presented in Chapter 3 to support self-healing composite service executions. In Section 4.1 we highlighted the main limitations of the transactional fault tolerance approach of Chapter 3 and the need of smarter composite service execution systems. We proposed a high-level definition of self-healing composite services in Section 4.2. In Section 4.3, we extended the architecture of Service Agents to provide them with more sophisticated autonomous behavior, we presented the main components of their architecture: the Service Agent API, Core, Autonomic Component, and Knowledge Base. In Section 4.5, we introduced QoS Manager component to coordinate additional QoS consumption requirements. Finally, we presented the algorithms related to this chapter in Section 4.6. Regarding the limitations of the approach presented this chapter, we may point out the same ones as for Chapter 3: the difficulty to do sound experimentation and real-world deployment. Another limitations is the fact that the QoS Manager for for summation/product QoS criteria presented in Section 4.5 is centralized; however, the QoS Manager is only needed in case of service replacements that need more summation/product QoS than expected. We leave the proposal of a distributed composite service QoS resource allocation approach for future work. Also as future work, we plan to study the automatic rule identification and self-configuration of execution parameters.

Chapter 5

Experimental Evaluation

Contents

5.1 Implementation Overview 114

5.2 Case Study 117

5.2.1 QoS dataset 118

5.2.2 The e-Health System 119

5.3 Results 125

5.3.1 Composite Service Behavior (*nt-sys*) 125

5.3.2 Experimental Comparison Between *nt-sys*, *tp-sys*, and *sh-sys* 128

5.3.3 Conclusions of Sections 5.3.2.1 and 5.3.2.2: *tp-sys* vs *sh-sys* 134

5.3.4 Self-healing Behavior 135

5.4 Summary of Experimental Evaluation 138

In this chapter, we present an implementation of our framework and evaluate it experimentally using a case study. For this case study, we include a description of its corresponding scenario and environment. We are interested in observing the case study running under three different systems: a non-fault tolerant, a pure transactional, and a self-healing approach. These three different systems are defined as follows:

- *nt-sys*: a system with no fault tolerance mechanisms. If a service fails, the system throws an exception and stops its execution.

- *tp-sys*: our pure transactional composite service execution system presented in Chapter 3. We call it *pure transactional* since it takes recovery decisions taking into account only the transactional properties of component services.
- *sh-sys*: our approach for self-healing composite service execution presented in Chapter 4. It makes decisions using information and rules contained in the Service Agent knowledge bases.

The rest of this chapter is organized as follows: Section 5.1 presents the implementation architecture of our system and the specific technologies used to build Service Agents; Section 5.2.1 describes the QoS dataset used to do the experimental evaluation presented in this chapter; Section 5.2.2 contains experimental observations using a case study; Finally, Section 5.4 concludes this experimental chapter.

5.1 Implementation Overview

We implemented our system as a Web application using Java EE 7. It follows the Web application architecture depicted in Figure 5.1. The user interface is presented as a Web page that allows the creation or uploading of composite services and their execution. Clients and servers maintain a full-duplex communication through a websocket. The communication among clients and the websocket is done by interchanging JSON messages. There is a Java thread for the Agent Coordinator and a Java thread for each Service Agent. The system was deployed in GlassFish Server Open Source Edition 4.1 (build 13).

The Service Agent-Service Agent and Service Agent-Agent Coordinator communication follows the asynchronous message passing paradigm. The agent coordinator and each Service Agent has a queue where messages can be posted by components participating in the current execution. We use the `LinkedBlockingQueue` Java implementation of a blocking queue. In this type of queue, if a Service Agent requests a message and its queue is empty, it will block and wait until there is something in the queue. Figure 5.2 shows a typical asynchronous message passing model where two agents communicate through a blocking queue.

Regarding the knowledge base inference engine, we use the Apache Jena framework[1]. This framework provides an API to manipulate RDF graphs and a generic

[1]https://jena.apache.org/

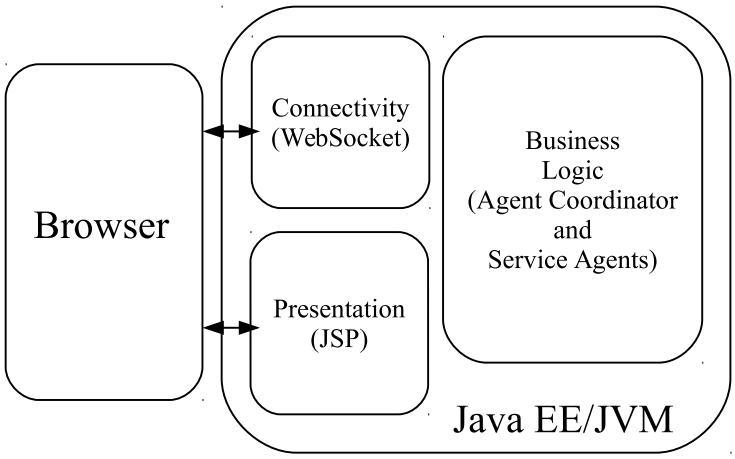


Figure 5.1: Web Application Implementation Architecture.

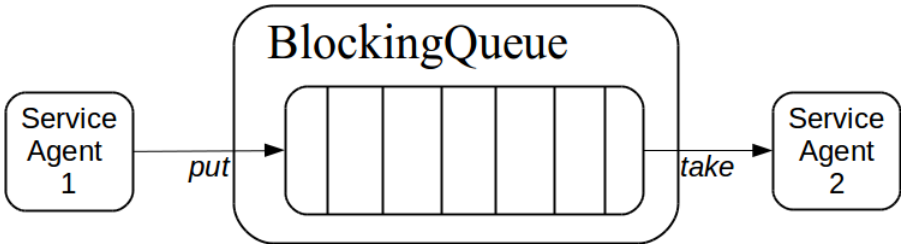


Figure 5.2: Asynchronous Message Passing.

rule reasoner[2] to infer logical consequences from a set of asserted facts. Although it is beyond the scope of this thesis, Jena is also compatible with the Semantic Web standards, allowing to share and reuse rules, and providing data interoperation [81].

Listing 5.1 provides an example of the usage of Jena. In the example, we first create a RDF model, which is a set of RDF statements. Then, a reasoner is created with a set of predefined inference rules. Finally, an inference model is created by attaching the instance of a reasoner to a set of RDF data. Additional entailments derived from a set of RDF data appear as additional RDF data in the inferred model. These models are loaded into memory in each Service Agent instance; we did not implement persistent storage.

Listing 5.1: Jena Example.

```
Model model = ModelFactory.createDefaultModel();
Reasoner reasoner = new GenericRuleReasoner(
      Rule.rulesFromURL("file:rules.txt"));
```

[2]https://jena.apache.org/documentation/inference/

```
reasoner.setDerivationLogging(true);
InfModel inf = ModelFactory.createInfModel(reasoner, model);
```

We use the Turtle[3] triple-like serialization to tell Service Agents the predefined inference rules. The predefined rules are the one presented in Section 4.4; however, they may be changed by users according to specific requirements. These rules are written in a text file `rules.txt` and loaded into the reasoner as showed in Listing 5.1. Listing 5.2 shows some simplified example rules written using the Turtle/N3 syntax. For example, the first rule tells that if the value a of certain variable d if less than a threshold *threshold*, then the state is *normal*; the second rule deduces a *broken* state if the value the same variable is greater than the threshold; the third rule deduces the action *compensate* if the state is *broken*; and the fourth rules deduces the action *continue* if the state is *normal*. Note that the third and fourth rules deduce actions based on the state inferred by the first two rules. Each time a triple is inferred, it is added to the RDF model.

Listing 5.2: Example Turtle/N3 Rules.

```
@prefix int: <http://www.w3.org/2001/XMLSchema#integer>.
@prefix : <http://jena.hpl.hp.com/prefix#>.
[ (?d int:value ?a) lessThan(?a,?threshold) ->
   (<http://example/State/> :state <http://example/State/Normal/>)]
[ (?d int:value ?a) greaterThan(?a,?threshold) ->
   (<http://example/State/> :state <http://example/State/Broken>)]
[ (<http://example/State/> :state <http://example/State/Broken>) ->
   (<http://example/Action/> :action
   <http://example/Action/Compensate>)]
[ (<http://example/State/> :state <http://example/State/Normal/>) ->
   (<http://example/Action/> :action
   <http://example/Action/Continue/>)]
```

Service Agents query their respective models using ARQ[4], which is a query engine for Jena that supports the SPARQL RDF Query language. For example, Listing 5.3 shows how to add a triple to a RDF graph using SPARQL. In this case, the subject is *time*, the predicate is *integervalue*, and the object is the actual value which comes from a Java variable called *time*. This way, Service Agents add initial and runtime information to their knowledge bases. Listing 5.4 shows the query used to retrieve the deduced action from the model. Note that SPARQL

[3]http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/
[4]https://jena.apache.org/documentation/query/

queries RDF graphs and it is the triples in the graph that matter; therefore, these queries are independent of the used serialization[5].

Listing 5.3: Example SPARQL INSERT.

```
INSERT DATA {<http://jena.hpl.hp.com/prefix#time>
    <http://www.w3.org/2001/XMLSchema#integervalue> time }
```

Listing 5.4: Example SPARQL SELECT.

```
SELECT ?action WHERE {<http://example/Action/> :action ?action }
```

Service Agents `Ask` and `Tell` interfaces are also implemented as SPARQL endpoints enabling humans or machines to communicate with Service Agents via the SPARQL language. For security reasons, in this work we suppose that Service Agents `Ask` and `Tell` interfaces are accessible only to components of the current context; that is, the corresponding Agent Coordinator and the Service Agents participating in the same composite service execution.

For our case study, we developed RESTful services using the Java API for RESTful Web Services (JAX-RS). They were also deployed in GlassFish. These services have a transactional property manually assigned; their execution time and failure probability were taken from the dataset we describe in the following section.

Finally, all the artifacts we implemented, including raw results and scripts to generated plots and tables, are available in a public Git repository[6].

5.2 Case Study

In this section, we present the case study we use to do the experimental evaluation of our approach. First, we describe the QoS dataset we use to simulate service behavior. Then, we propose the fictional e-Health composite service as case study.

[5]https://jena.apache.org/tutorials/sparql_data.html
[6]https://bitbucket.org/rafaelangarita/

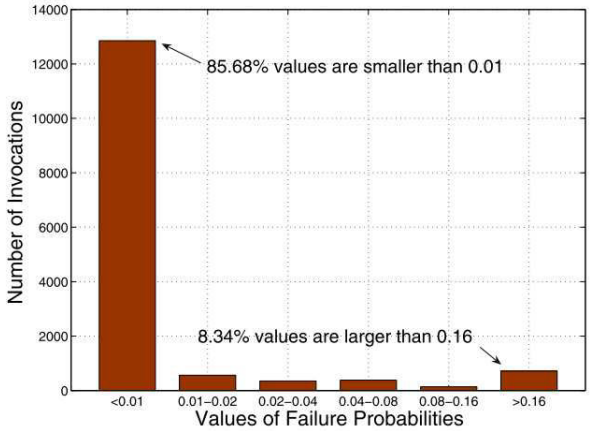


Figure 5.3: Distribution of failure probabilities from [103].

5.2.1 QoS dataset

We have said that services participating in our case study are RESTful services created by us using JAX-RS; however, we still need to provide them with their corresponding behavior. The behavior of a service can be seen as its perceived QoS; therefore, to make our simulations closer to true experimentation, we use the real world services QoS results presented in the WS-DREAM dataset [103]. In their work, Zheng et al. conduct large scale evaluations on real services to measure failure probabilities and response time.

In [103], the evaluation of failure probabilities was done on 100 randomly selected services from a set of 13,108 services. The 100 services were invoked for about 100 times by 150 users distributed in 24 countries. In total, 1,542,884 invocations were collected. Figure 5.3 shows the value distribution of the computed failure probabilities. We can see that 85.68% of the failure probabilities are smaller than 1%, but 8.34% are higher than 16%. Table 5.1 shows the statistics of the failure probability evaluation.

Regarding services response times, Zheng et al. performed 1,974,674 real world service invocations. The invocations were done by 339 users distributed over 30 countries. 5,825 real services from 73 countries were used. Table 5.2 shows the statistics for the response time evaluation. Figure 5.4 shows the overall response time of services and users.

We consider two QoS criteria: execution time, and availability. Execution times and availabilities are randomly generated according to the means and standard deviations previously described in this section and presented in [103]. Figure 5.5

Statistics	Values
Num. of service invocations	1,542,884
Num. of users	150
Num. of services	100
Num. of user countries	24
Num. of service countries	22
Range of failure probability	0-100%
Mean of failure probability	4.05%
Standard deviation of failure probability	17.32%

Table 5.1: Statistics of the dataset 1 from [103].

Statistics	Values
Num. of service invocations	1,974,675
Num. of users	339
Num. of services	5,825
Num. of user countries	30
Num. of service countries	73
Mean of response time	1.43 s
Standard deviation of response time	31.9 s
Mean of throughput	102.86 kbps
Standard deviation of throughput	531.85 kbps

Table 5.2: Statistics of the dataset 2 from [103].

illustrates the distribution of the failure probabilities for 10,000 services. The distribution of the response times of the 10,000 services can be observed in Figure 5.6.

5.2.2 The e-Health System

Figure 5.7 illustrates a Colored Petri net representing a fictional e-Health application we created for this evaluation. This application is composed by 9 services: *Sugar Implant*, *Vital Signs Implant*, *Sugar Analysis*, *Vital Signs Analysis*, *Diagnoser*, *Call Emergency*, *Notify Contact*, *Notify Doctor*, and *Display Message*.

The *Sugar Implant* and *Vital Signs Implant* services receive information about a patient wearing smart devices. Each of them process this information and send it to their respective analysis services, which send their conclusions to the

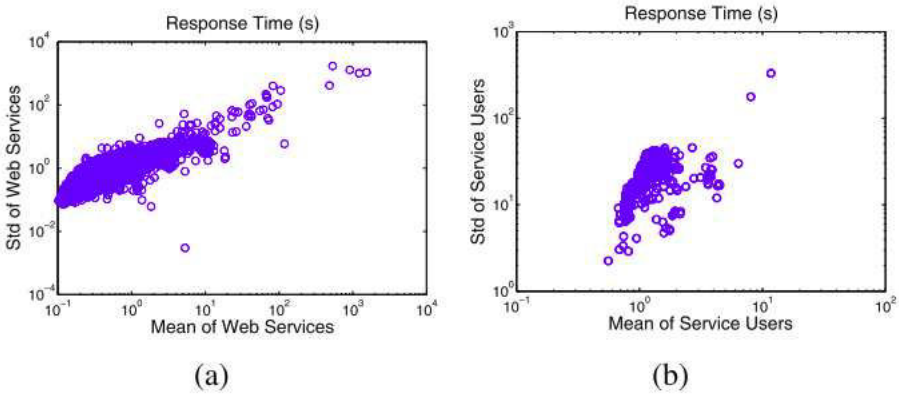


Figure 5.4: Overall response time from [103].

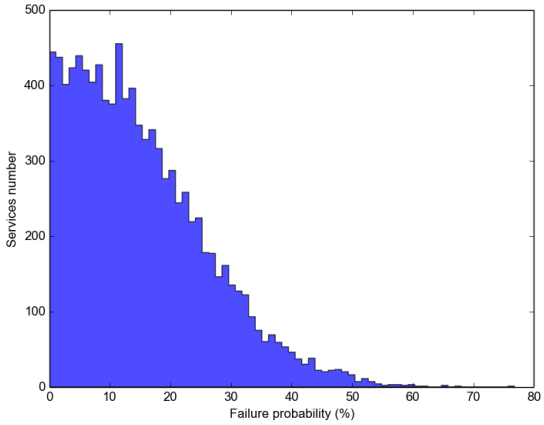


Figure 5.5: Distribution of generated failure probability for 10000 services.

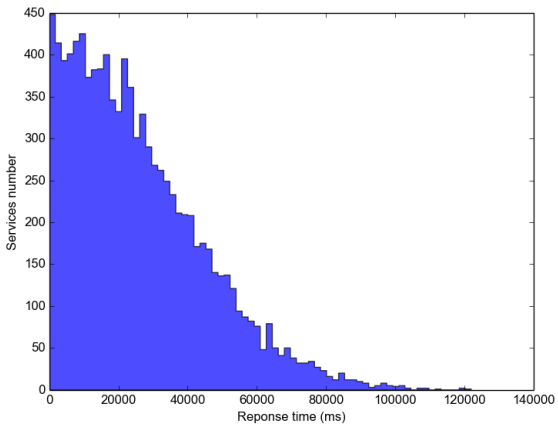


Figure 5.6: Distribution of generated response time for 10000 services.

service	*time (ms)*	*availability*	*tp*
SugarImplant	12263.33	0.77	*cr*
VitalSignsImplant	51712.56	0.7	*cr*
SugarAnalysis	27403.77	0.96	*c*
VitalSignsAnalysis	10672.81	0.91	*cr*
Diagnoser	11671.86	0.87	*c*
CallEmergency	16123.66	0.90	*cr*
NotifyContact	1916.14	0.65	*cr*
NotifyDoctor	34228.88	0.93	*c*
DisplayMessage	21320.73	0.98	*c*
Composition	107287.06	0.22	*c*

Table 5.3: Case Study QoS Values and Transactional Properties.

Diagnoser service. The *Diagnoser* service sends its results to take the appropriate actions.

This system is considered as critical since patient lives depend on it. Let us suppose for this experience that experts do not want the e-Health application to exceed a certain amount of time during its execution since it can be dangerous to patients: if the system execution takes *too much time*, then it should be stopped and call a nurse immediately through an emergency protocol. Therefore, the system is restricted by an execution time threshold θ_{time}. We suppose that the emergency protocol consists on invoking a less sophisticated service which calls or sends a message to a nurse directly with a high probability of success.

Table 5.3 shows the QoS values and transactional properties of each service in the composite service showed in Figure 5.7. The last line of the table shows the global values of the composite service. In particular, we highlight the low global availability, which is given by the product of individual availabilities:

$$\begin{aligned} \circ_{availability} &= \prod_{1}^{9} \gamma_{availability} \\ &= 0.22 \end{aligned}$$

Figure 5.8 shows the execution time knowledge computed by the Agent Coordinator (Def. 4.3.5, 4.3.8, and 4.3.9). For example, if we look at the *VitalSignsImplant* Service Agent, its knowledge about execution time is the following:

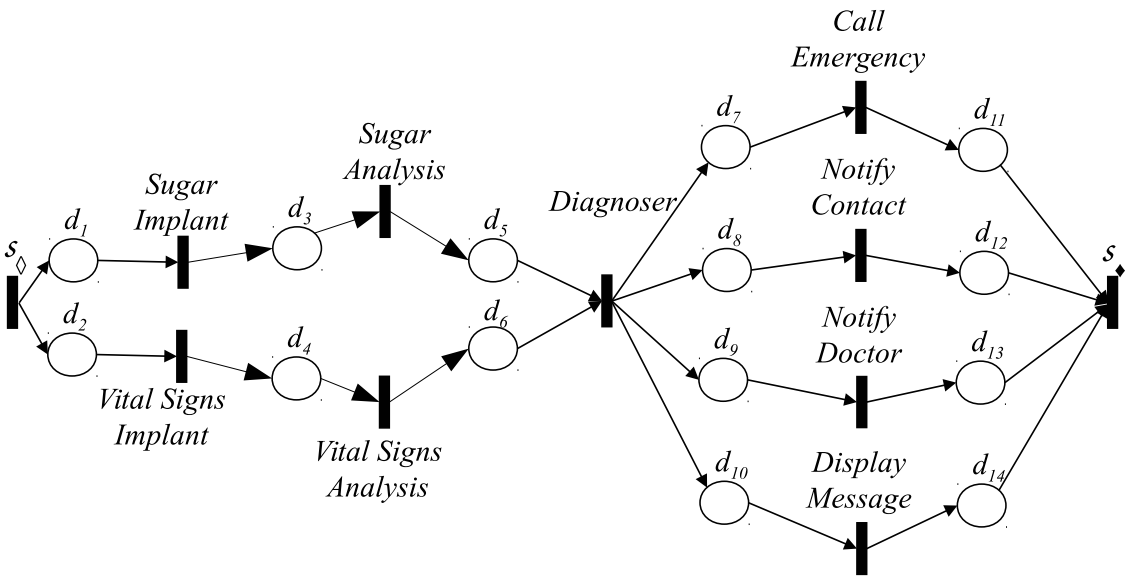


Figure 5.7: Case Study: e-Health Application.

$$\gamma_{VitalSignsImplant^{\approx}_{time}} = 51712.56ms$$
$$\gamma_{VitalSignsImplant^{\approx}_{firing}} = 0.0ms$$
$$\gamma_{VitalSignsImplant^{\approx}_{remaining}} = 56573.54ms$$
$$\gamma_{VitalSignsImplant^{+\approx}_{time}} = 0.0ms$$

If we take the *SugarImplant* service, which is parallel to *VitalSignsImplant*, we can see that:

$$\gamma_{SugarImplant^{\approx}_{time}} = 12263.33ms$$
$$\gamma_{SugarImplant^{\approx}_{firing}} = 0.0ms$$
$$\gamma_{SugarImplant^{\approx}_{remaining}} = 73304.51ms$$
$$\gamma_{SugarImplant^{+\approx}_{free}} = 22718.26ms$$

Note that the extra available time of $VitalSignsImplant$ is $\gamma_{VitalSignsImplant^{+\approx}_{time}} = 0.0ms$. This means that *VitalSignsImplant* is in the critical path of the composite service, then its execution time can directly affect the global execution time of the composite service. *SugarImplant* can use some extra execution time without

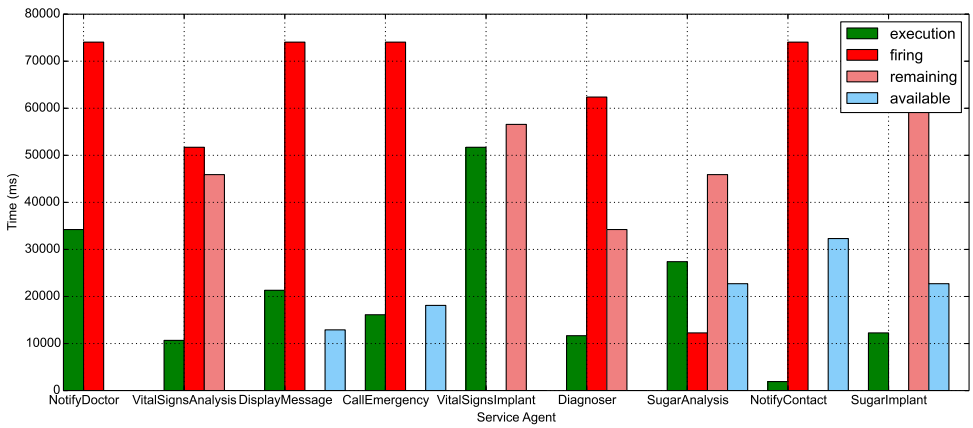


Figure 5.8: Composite Service Knowledge: Time.

affecting the whole composite service. The complete set of Service Agents in the critical path is the following (Def. 4.3.5):

$$CriticalPath = \{VitalSignsImplant,\ VitalSignsAnalysis;\\ Diagnoser,\ NotifyDoctor\}$$

Remember that execution time knowledge is a set of execution time estimations which can change during the execution of the composite service; however, for Service Agents receiving the composite service inputs we have that:

$$\forall \gamma \in (\gamma_{\Diamond}^{\bullet})^{\bullet},\ \gamma_{firing}^{\approx} = \gamma_{firing} = 0$$

The computed progress (Def. 4.3.17) for each Service Agent is depicted in Figure 5.9. This progress knowledge was computed by the Agent Coordinator using $\omega_{predecessors} = \omega_{d-outputs} = 0.5$. We can see that $\gamma_{progress} < 10\%$ for $VitalSignsAnalysis$, $VitalSignsImplant$, $SugarAnalysis$, and $SugarImplant$ Service Agents; $\gamma_{progress} > 60\%$ for the rest of the Service Agents excepting $Diagnoser$, which progress is about 22%.

Note that composite service knowledge about the execution time and the composite service progress are only taken into account by *sh-sys*. *tp-sys* only considers transactional properties.

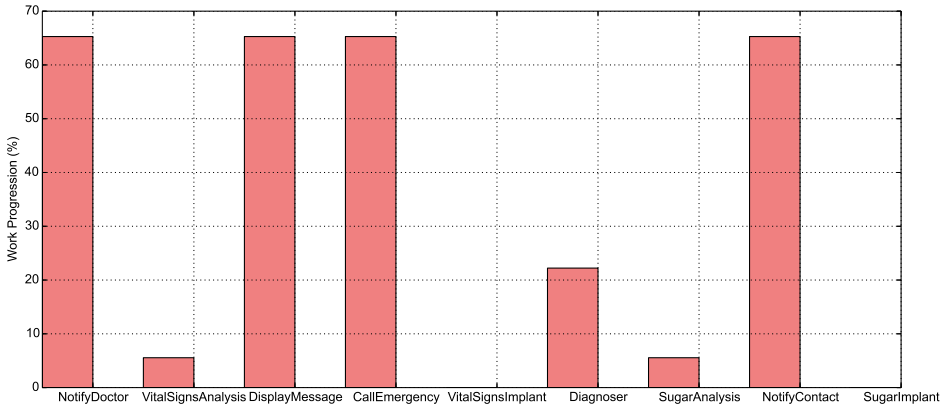


Figure 5.9: Composite Service Knowledge: Progress.

	Value
θ_{time} (Def. 4.3.23)	5%
$\theta_{progress}$ (Def. 4.3.27)	70%
$\theta_{replication}$ (Def. 4.3.26)	0.5
$\omega_{predecessors}$ (Def. 4.3.17)	0.5
$\omega_{d-outputs}$ (Def. 4.3.17)	0.5

Table 5.4: *sh-sys* execution Parameters.

Table 5.4 shows the values of the execution parameters used for this experience. We can see that the execution time threshold (θ_{time}) is only 5% of the global estimated execution time.

The fault tolerance mechanisms considered in this evaluation are: backward recovery through compensation (Section 3.1) for *tp-sys* and *sh-sys*; forward recovery by service retrying (Section 3.2.1) for *tp-sys* and *sh-sys*, and replication (Section 4.4) as a proactive mechanism for *sh-sys*. We consider the following for the evaluations presented in this chapter:

- there are no replacement services. Forward recovery is done only by service retrying. In fact, results with service replacement are the same as with service retry. The only different is that other QoS may change with service replacement, such as execution price;
- we do not activate the checkpointing option. We can think of it as an alternative to compensated executions showed in these results;

- the actual service execution time is the same as its estimated execution time. Service either fail or succeed, but their execution time is equal to their estimated value.

5.3 Results

In this section, we present the most relevant experimental results using the case study proposed in Section 5.2.2. First, in Section 5.3.1, we propose an analysis of our case study using the non-fault tolerant system, *nt-sys*. In Section 5.3.2, we present an experimental comparison between *nt-sys*, *tp-sys*, and *sh-sys*. Finally, in Section 5.3.4, we present some experimental observations specific to *sh-sys*.

5.3.1 Composite Service Behavior (*nt-sys*)

Before continuing with the actual evaluation of our approach, we present the observed behavior of our case study. This evaluation is done by executing the composite service using the *nt-sys*; that is, our basic execution system without fault tolerance capabilities. We remind that *nt-sys* does not take into account the transactional properties of the e-Health application, and services may fail according to their availability independently of their transactional property. We executed the composite service illustrated in Figure 5.7 100 times.

Figure 5.10 illustrates the ratio between successful and failed executions. We can see that only 17% of the execution finished without any failure, which is not surprising given the availability of the composite service.

Figure 5.11 shows the ratio among the number of failures in failed executions. 69% of the failed executions had one failure; 14% of the failed executions had two failures. Only 17% of the total executions were successful with no failures.

Figure 5.12 shows a scatter plot of failure occurrences for 100 executions of our composite service under *nt-sys*. In our simulations, service execution times are the same than their estimated execution time even if a failure occurs; therefore, when a service fails, it has already consumed its estimated execution time. For example, we can see that: a failure occurred between 0 ms and 20,000 ms, and between 40,000 ms and 60,000 ms in several executions, but few failures occurred after 100,000 ms.

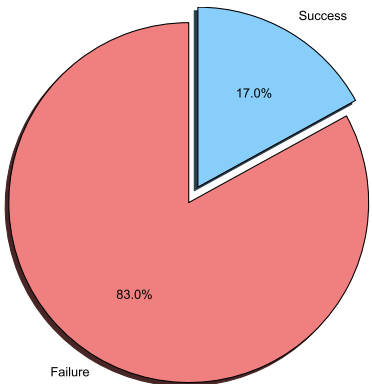


Figure 5.10: Failure percentage (*nt-sys*).

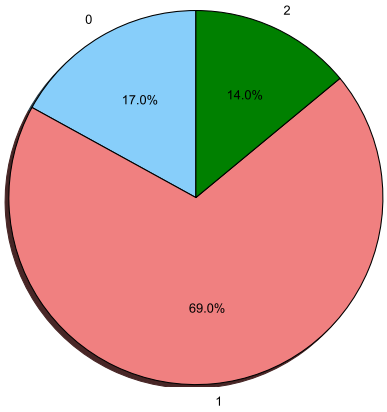


Figure 5.11: Failure percentage by failure number (*nt-sys*).

We have also observed which services failed and when they failed (Table 5.5). Most of the times, the composite service failed because of the malfunction of the services *SugarImplant* and *VitalSignsImplant*. This can be explained by the fact that both services have relatively low availability and they are at the beginning of the composite service, so if they fail, their successors will not be executed.

Some services may be executed successfully despite the failure of other services in the same composite service execution. If this happens, we say that the system is in an inconsistent state. Table 5.6 shows measurements of system inconsistency. We may see that the maximum number of failed services was 2, while the maximum number of successful services in a failed execution was 8. We may also use this metric as a measure of lost work; for example, the worst case was when

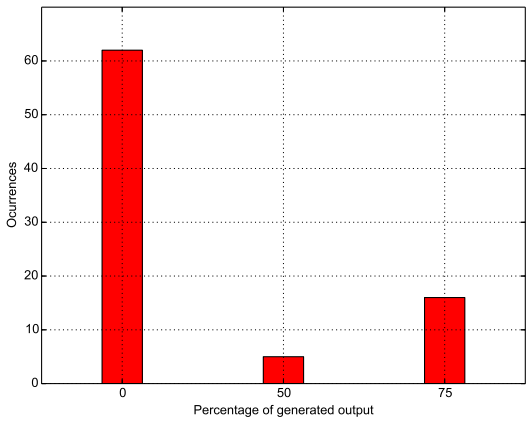


Figure 5.12: Scatter plot of failure occurrences for 100 executions (*nt-sys*).

service	*failure (%)*
VitalSignsAnalysis	6.02409638554
CallEmergency	7.22891566265
SugarAnalysis	4.81927710843
NotifyContact	18.0722891566
SugarImplant	26.5060240964
VitalSignsImplant	42.1686746988
Diagnoser	6.02409638554
NotifyDoctor	6.02409638554

Table 5.5: Failed services and their failure percentage (*nt-sys*).

8 services out of 9 were executed for nothing.

The last observation is the amount of generated outputs during failed executions. We consider this values of generated outputs as a way to measure lost work along with the number of successful services in a failed execution. We can see in Figure 5.13 that, for most of the failed executions, none of the composite service output was generated; for around 5% of the executions half of the output was generated; and for around 17% of the executions 75% of the output was generated. We consider that generated composite service outputs are useless after a failed execution due to system inconsistency.

	Failed	*Success*
Max	2	8
Min	1	1
Mean	1.1686746988	3.53012048193

Table 5.6: System inconsistency (*nt-sys*).

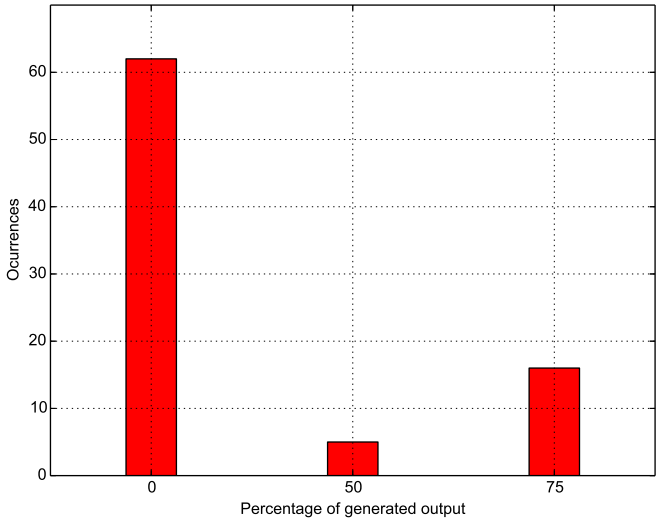


Figure 5.13: Generated outputs during failed execution (*nt-sys*).

5.3.2 Experimental Comparison Between *nt-sys*, *tp-sys*, and *sh-sys*

In this section, we include the most relevant experimental comparison between *nt-sys*, *tp-sys*, and *sh-sys*. Since *nt-sys* is not really an approach, we use it as an illustration of what would happen if we did not implement fault tolerance mechanisms for composite services. We do the experimental evaluation of the following two metrics:

1) Composite service execution time: we have chosen the analysis of the composite service execution time to illustrated the QoS-awareness. A similar analysis follows for other QoS criteria.

2) Resource wastage: successfully executed services in failed executions, and successfully executed and compensated services.

5.3.2.1 Execution Time Analysis

We analyze and compare the observed execution times of our three systems: *nt-sys*, *tp-sys*, and *sh-sys*. Table 5.7 presents the following information about execution time:

- Mean Time (ms): it is the mean time of executions with no failures; that is:

$$MeanTime = mean(\circ_{time}^{\neg failure})$$

- Overhead (%): it is the execution overhead relative to the global estimated execution time. It is calculated as follows:

$$overhead = \frac{MeanTime - \circ_{time}^{\approx} * 100}{\circ_{time}^{\approx}}$$

- $BROKEN_{time}$: it is the percentage of executions with exceeding QoS; that is, that satisfy the rule $\mathcal{R}_{QoS^2}$ of Section 4.4.1. In this evaluation, we evaluate only execution time; thus, we count executions which satisfy:

$$BROKEN_{time} = \frac{(count(\circ_{time})|\ \circ_{time} > \circ_{time}^{\approx}) * 100}{count(\circ_{time})}$$

  where $count(\circ_{time})|\ \circ_{time} > \circ_{time}^{\approx}$ is the number of executions with exceeding execution time, and $count(\circ_{time})$ is the total number of executions.

- Max Time (ms): it is the maximum time among executions with no failures:

$$MaxTime = max(\circ_{time}^{\neg failure})$$

- Mean Failed Time (ms): it is the mean time of executions with failures:

$$MeanFailedTime = mean(\circ_{time}^{failure})$$

	nt-sys	*tp-sys*	*sh-sys*
Overhead (%)	0.0060911619717	0.00711834769777	0.00867147226917
$BROKEN_{time}$ (%)	0.0	30.0	0.0
Mean Time (ms)	108286.11	134103.61039	108320.886731
Max Time (ms)	108286.11	263423.79	110094.5
Mean Failed Time (ms)	70201.3518072	90744.0434783	72600.0416667
Max Time Failed (ms)	108286.11	95387.0	95388.0
Mean Compensation Time (ms)	n/a	166241.559565	103376.655625

Table 5.7: Execution Times.

- Max Time Failed (ms): it is the maximum time among executions with failures:

$$MaxFailedTime = max(\circ_{time}^{failure})$$

- Mean Compensation Time (ms): it is the mean time of compensated executions:

$$MaxCompensationTime = max(\circ_{time}^{compensation})$$

  where $\circ_{time}^{compensation}$ refers to the time of compensated executions.

Table 5.7 shows the different execution times under the three evaluated systems. We can see that the *overhead* of the three systems is less than 0.009%. The overhead of *sh-sys* is higher but close to the overheads of *nt-sys* and *tp-sys*. In conclusion, the overhead incurred by adding self-healing behavior is acceptable in relation to the global execution time, to the overheads of *tp-sys*, and to the overhead of an execution system with no fault tolerance, *nt-sys*.

Regarding $BROKEN_{time}$, *nt-sys* has never exceeded the execution time, since successful services do not take more time than expected and failed executions stop at the first failure. *tp-sys* exceeds the execution time by 30% due to successful composite service executions using retry. *sh-sys* has never exceeded the execution time due to its QoS-awareness capabilities.

The *mean times* for *nt-sys* and *sh-sys* are close to the composite service estimated execution time $\circ_{time}^{\approx}$, which is 107287.06, but the mean time for *tp-sys* is

higher. This is due to the same reason why $BROKEN_{time}$ is higher for *tp-sys*; that is, global execution time degradation caused by service retry.

Again, for the *max time*, the values of *nt-sys* and *sh-sys* are again close to $\circ^{\approx}_{time}$; however, the max time for *tp-sys* is more than the double of the max times of *nt-sys* and *sh-sys*. This happens for the same reason exposed for the $BROKEN_{time}$ and *mean time* measurements, and it illustrates the worst case for *tp-sys* execution time using our study case.

The *mean failed time* for *nt-sys* represents the execution time when the first service failed, and exception was thrown, and the whole execution was stopped. For *tp-sys* and *sh-sys*, it is the time when the system decided to compensate. We can see that mean failed time for *nt-sys* is close of the mean failed time for *sh-sys*; however, *tp-sys* has a higher mean failed time due to the composite executions that were compensated after doing forward recovery. This measurement shows that *sh-sys* keeps the execution time low for failed executions in comparison to *tp-sys*, which wastes resources by compensating after doing forward recovery.

The *max time failed* is similar for *tp-sys* and *sh-sys*, while it is slightly higher for *tp-sys*. In all cases, it reflects the situation when irreparable failures occurred after many of the component services had been successfully executed, in many cases, using forward recovery.

The final time related measurement regards the compensation time, which does not exist for *nt-sys*, since it does not provide fault tolerance. The *tp-sys mean compensation time* is 166241.559565 ms, against the *sh-sys mean compensation time* of 103376.655625 ms. This measurement also reflects the resource wasting by *tp-sys*.

5.3.2.2 Resource Wastage

We define the resource wastage in a composite service execution as the services successfully executed *for nothing*; that is, wasted service executions. In the case of *nt-sys*, it refers to the successfully executed services in failed executions. For *tp-sys* and *sh-sys*, it refers to successfully executed and compensated services. A consequence of wasted service executions is the *lost outputs*. For, *nt-sys* the lost outputs are the produced composite service outputs in a failed execution, while for *tp-sys* and *sh-sys* are the produced composite service outputs in a compensated execution.

	nt-sys	*tp-sys*	*sh-sys*
Total Failure Count	97	125	101
Max Failure Count	2	3	3
Successful Executions (%)	17.0	77.0	52.0
Failed Executions (%)	83.0	23.0	14.0
Stopped Executions (%)	n/a	n/a	36.0
Compensated Services Mean	n/a	4	2
Max Compensated Services	n/a	8	8
Lost Outputs (%)	17.4698795181	2.17391304348	0.520833333333

Table 5.8: Resource Wastage.

Table 5.8 shows information related to the resource wastage under *nt-sys*, *tp-sys*, and *sh-sys*. This information will help in the understanding of the results presented in Section 5.3.2.1. The information showed in Table 5.8 is the following:

- Total Failure Count: it refers to the count of service failures of all composite service executions. Some executions may have been free of failures, while other executions may have had one or more failures.
- Max Failure Count: it reflects of the maximum number of failures occurred during a single composite service execution.
- Successful Executions (%): it is the percentage of successful execution out of all executions.
- Failed Executions (%): it is the percentage of failed execution out of all executions.
- Stopped Executions (%): it is the percentage of stopped execution out of all executions.
- Compensated Services Mean: it is the mean of the compensated service number in compensated executions.
- Max Compensated Services: it is the maximum number of compensated services during a single compensated execution.
- Lost Outputs (%): it is the mean of the percentage of generated outputs during failed executions.

In the *total failure count* column of Table 5.8, we can see that *nt-sys* produced the lower number of failures, followed closely by *sh-sys*. *tp-sys* produced about 25% more failures than the two other systems due to service retry. The *max failure count* remains about the same for the three systems.

Regarding the *successful execution percentage*, *nt-sys* had only 17% of successful executions expected, since *nt-sys* does not implement fault tolerance mechanisms and $\circ^{\approx}_{availability} = 0.22$. *tp-sys* had 77% of successful executions by using service retry, and *sh-sys* 52% of successful executions also by using service retry.

The *failed executions percentage* for *nt-sys* and *tp-sys* are the complement of their corresponding percentage of successful executions; that is, 83% for *nt-sys* and 23% for *tp-sys*. An important observation is that this is not the case for *sh-sys*, for which the percentage of failed executions is only 14%. The missing executions for *sh-sys* were stopped executions. Table 5.8 shows 36% of stopped executions for *sh-sys*; this is a non-existing concept for *nt-sys* and *tp-sys* since *nt-sys* does not implement any fault tolerance mechanism, and *tp-sys* only considers stopping an execution when forward recovery is not possible. *sh-sys* stops executions that do not satisfy $\theta_{time} = 5\%$, which is part of the execution parameters of Table 5.4.

The *compensated service mean* was 4 for *tp-sys* and 2 for *sh-sys*. This also can be seen as if *tp-sys* wasted more resources than *sh-sys*. For example, 4 compensated services means 4 successful executions and 4 executions to compensated them; that is 8 compensation related executions of *tp-sys* against 4 of *sh-sys*. For both systems, the *maximum number of compensated services* was 8.

Another metric that gives an idea about resource wasting is the *lost output percentage*. *nt-sys* has a high lost output percentage of 17.46%. It also represents system inconsistency for *nt-sys*, since the composite service produced some outputs before failing, while it does not provide a compensation mechanism. *tp-sys* produced a mean of 2.17% of the outputs during failing executions; while *sh-sys* only produced 0.52% of outputs, which is still better than the 2.17% of *tp-sys*. Both *tp-sys* and *sh-sys* left the system in a consistent state by compensating successful executed services.

5.3.3 Conclusions of Sections 5.3.2.1 and 5.3.2.2: *tp-sys* vs *sh-sys*

Experimental evaluation presented in previous sections illustrates an example of why it is necessary to provide fault tolerance for large, distributed, and heterogeneous systems running in unreliable environments such as the Internet. In Sections 5.3.2.1 and 5.3.2.2 we haven showed the observed behavior of *nt-sys*, which is a basic composite service execution system without fault tolerance. Besides the low number of successful executions, the main problems of *nt-sys* are the inconsistent system states and the wasted resources after failed executions.

Here, the question is to compare our systems providing fault tolerance: *tp-sys* and *sh-sys*. The most important general property both systems have in common is the transactional-based fault tolerance. Both systems guarantee the system consistency even in the presence of failures.

The main difference between *tp-sys* and *sh-sys* is that *tp-sys* takes decisions based only on transactional properties, triggered only when a service fails, and it is not aware of the composite service QoS. We described the possible problems of this approach in Section 4.1 and illustrated the transactional-based decision making model in Figure 4.1. The main exposed disadvantaged for *tp-sys* are QoS degradation and lost work.

Section 5.3.2.1 showed that *tp-sys* finished with QoS degradation 30% of the times, while *sh-sys* never had QoS degradation. In the worst case, *tp-sys* finished with $\circ_{time} = 263423.79$, which is more than the double of the expected $\circ^{\approx}_{time} = 107287.06$. Regarding lost work, Section 5.3.2.2 shows that *tp-sys* incurs in greater lost of work than *sh-sys* due to higher number of successful execution, output generation, and compensation of components services of failed executions.

In conclusion, *tp-sys* provides a transactional-based deep-seated notion of acceptable behavior to guarantee system consistency; however, *tp-sys* bases its decisions only on its transactional properties, so it neglects other requirements such as QoS. To solve this problem, *sh-sys* builds on top of *tp-sys* to provide fault tolerance for composite services by extending Service Agents with a knowledge base capable of taking decisions based on QoS and other user preferences.

	Normal	*Degraded*	*Broken*
Normal	76.7189384801	9.04704463209	5.79010856454
Degraded	2.29191797346	5.9107358263	0.241254523522
Broken	0.0	0.0	0.0

Table 5.9: Self-healing State Transitions (%).

5.3.4 Self-healing Behavior

The rest of this experimental evaluation corresponds to the observation of aspects that are specific to the self-healing behavior; that is, we focus on the study of *sh-sys*. We made the following observations about *sh-sys*:

- changes in the composite service self-healing states introduced in Section 4.2;
- actions taken from the different self-healing states, which were illustrated in Figure 4.4.
- actions taken vs default transactional actions. Default transactional actions are equivalent to the behavior of *tp-sys*;
- actions taken by monitoring stage; that is, at which moment of their life-cycle Service Agents deduced which actions.

5.3.4.1 Self-healing State Transitions

The transition among *normal*, *degraded*, and *broken* self-healing states are showed in Table 5.9. States on the left represent the initial state, while states on the top are the states to which the transition was made. For example, for 76.71% of the times the self-healing state was *normal* and no transition was made. 2.29% of the transitions were from *degraded* to *normal* state. Finally, even though Table 5.9 shows that there was no implicit transition from the *broken* state; however, we consider that after compensation, the system goes always from *broken* to *normal* state. Conceptually, going to a *normal* state through compensation means that the system was left in a consistent state close to the one it had before the failed execution of a composite service.

	Continue	*Retry*	*Replicate*	*Compensate*
Normal	38.102189781	0.0	61.897810219	0.0
Degraded	4.25531914894	42.5531914894	53.1914893617	0.0
Broken	0.0	0.0	0.0	100.0

Table 5.10: Self-healing States to Recovery Plans (%).

5.3.4.2 Actions By Self-healing State

We can see which actions were taken from each of the self-healing states in Table 5.10. From the *normal* state, 38.10% of the times the action was *continue*; 61.89% of the times the action was *replicate*. *sh-sys* replicated services in the critical path or services with no time to retry or replace in case of failure. Of course, there are no failures in the *normal* state, so *retry* and *compensate* are not taking into account in this state. From the *degraded* state, 4.25% of the times the selected action was *continue*, 42.55% of the times it was *retry* after a service failure, and 53.19% of the times it was *compensate* because a failed service was not retriable or due to QoS violation. The only action considered from the *broken* state was *compensate*.

5.3.4.3 Self-healing vs Transactional Choices

We also observed the action taken by *sh-sys* vs the default actions (Table 5.11). The default actions are the recovery mechanism of the pure transactional system. Evidently, the pure transactional system does not take actions if there are no failures, so 100.0% of the times the self-healing system chose *continue*, the transactional system also chose *continue*. 100.0% of the times the self-healing system chose *retry*, the transactional system also chose *retry* since the only option is if the service is retriable. The *replicate* and *compensate* actions are more interesting. 93.67% of the times the self-healing system chose *replicate*, the transactional system did not take any action. This is because replication does not exist in the transactional system, while the self-healing system can use it as a preventive action.

	Continue	*Retry*	*Replicate*	*Compensate*
Continue	100.0	0.0	0.0	0.0
Retry	0.0	100.0	0.0	0.0
Replicate	93.6708860759	6.32911392405	0.0	0.0
Compensate	0.0	72.0	0.0	28.0

Table 5.11: Self-healing Recovery Plans vs Default Recovery Plans (%).

	Continue	*Retry*	*Replicate*	*Compensate*
Fireable	32.5227963526	0.0	67.4772036474	0.0
Failure	0.0	33.3333333333	25.0	41.6666666667
Fix	100.0	0.0	0.0	0.0

Table 5.12: Self-healing Recovery Plans by Monitoring Stage (%).

5.3.4.4 Self-healing Actions by Monitoring Stage

Table 5.12 shows the deduced actions by three monitoring stages: fireable, failure, and fix. The fireable monitoring is done before invoking a fireable service (Def 3.1.5); the failure monitoring is done right after a service failure; and the fix monitoring is done after a service failure is fixed.

When services were fireable, 32.52% of the times *sh-sys* chose to continue the execution as normal, without taking any special action. 67.47% of the times *sh-sys* chose to replicate to try to maintain the required QoS which, in this case, it is the execution time limited by its corresponding threshold (Table 5.4).

When services failed, 33.33% of the times *sh-sys* chose to perform retry, since the QoS stayed into the acceptable values. 25.0% of the times *sh-sys* chose to replicate to avoid a new service failure, and therefore, QoS violation. 41.66% of the times *sh-sys* chose to compensate could have been because the failed service was not retriable or to avoid QoS violation.

Finally, 100% of the times a failure was fixed *sh-sys* chose to continue.

5.4 Summary of Experimental Evaluation

In this chapter, we reported on our experience of implementing and executing a case study composite service. We compared the behavior of three different systems: *nt-sys*, a system with no fault tolerance mechanisms; *tp-sys*, the pure transactional composite service execution system presented in Chapter 3; and *sh-sys*, our approach for self-healing composite service execution presented in Chapter 4.

In Section 5.1, we presented the implementation architecture of our system and the specific technologies used to build Service Agents. We implemented our system as a Web application using Java EE 7, Service Agent knowledge is stored in a RDF graph and queried using SPARQL. We developed RESTful services using the Java API for RESTful Web Services. Finally, our system was deployed in GlassFish 4.1. In Section 5.2.1 and Section 5.2.2, we described the real QoS dataset used to perform the experimental evaluation in this chapter and proposed the fictional e-heath system as case study. In Section 5.2.2, we reported the experimental observations under *nt-sys*, *tp-sys*, and *sh-sys*.

In conclusion, the experimental evaluation presented in this chapter shows: (i), the need of providing fault tolerance mechanisms for composite services; (ii), how our approach of Chapter 3 handles composite service failures using transactional properties; and (iii), how our self-healing approach of Chapter 4 builds on top of our transactional approach to provide QoS-aware decision making. The evaluation presented in this chapter suggests that combining transactional properties with self-healing capabilities leads to smarter execution systems with the ability to handle higher-level requirements for composite service executions with minimal human intervention.

Chapter 6

Fault tolerance and self-healing composite service execution: an state of the art

Contents

6.1 Fault tolerance for composite services 140

6.1.1 Transactional Properties-based Approaches 141

6.1.2 Redundancy and Design Diversity-based Approaches . 142

6.1.3 Exception Handling-based Approaches 143

6.1.4 Prediction and Optimization Approaches 146

6.2 Self-healing execution of composite services 147

6.2.1 BPEL-based approaches 147

6.2.2 Non-BPEL-based approaches 148

6.3 Discussion 150

In this chapter, we propose a review of existing approaches to support the reliable execution of composite services. The selected works can be classified as fault tolerance research for composite services (Section 6.1), and as self-healing research for composite services (Section 6.2). In many cases, self-healing research focuses on QoS monitoring and pre- and post-conditions satisfaction, and makes explicit reference to the *monitor, diagnose, recover* loop of autonomic computing [71]. Fault tolerance research may focus less on monitoring and more on the

definition of the actual recovery mechanisms; however, the research classified into these two categories may sometimes overlap.

The search for related work was used using the following databases:

1) ACM Digital Library;
2) IEEE Electronic Library;
3) SpringerLink;
4) Elsevier.

From the results, we selected peer-reviewed journals articles, conference proceedings, and book chapters. We searched for work published between the years 2005 and 2015, and we selected the ones which provide direct evidence to the research question presented in Section 1.2 and were written in English. Fault tolerance or self-healing research concerning Service Oriented Architectures but not focused on services was excluded. Also, papers centered on providing fault tolerance only at composition/design-time, and not at runtime, were excluded. Finally, we submitted variants of the following search string to the research databases:

> *(fault-tolerance OR fault-tolerant OR fault tolerance OR fault tolerant OR self-healing OR adaptive OR reliable OR dynamic) AND (composite Web service OR composite service OR composite Web service execution OR composite service execution)*

6.1 Fault tolerance for composite services

In some works, transactional properties of component services are considered to ensure the system consistency even in presence of failures [24, 25, 51]. Some works focus on implementing redundancy and design diversity techniques such as replication [8, 32, 38, 57, 64, 104]. Exception handling is also a used technique for managing failures in composite services [22, 78, 84, 85]. Finally, other approaches implement prediction and optimization techniques [89] for improving the reliability of composite services.

6.1.1 Transactional Properties-based Approaches

Cardinale and Rukoz [25] propose a fault tolerance execution approach based on the transactional composite service model presented in [34] (we a have discussed this transactional model in Section 2.2.2). Composite services and their execution processes are modeled using Colored Petri nets. The proposed recovery techniques are: backward recovery by compensation, forward recovery by retry or replacement, and semantic recovery. The usage of transactional properties guarantee the consistency of the system event in the presence of failures by applying recovery mechanisms in an automatic way. This way, developers or automatic composition systems only have to care about building correct transactional composite services since the fault tolerance execution is automatically done. When a service fails, if it is retriable, it is retried until it finishes successfully; if it is not retriable, it can be substituted by another service that satisfies its functional and transactional requirements; if forward recovery is not possible, compensation is chosen as the recovery mechanism. In contrast, our approach provides checkpointing as an alternative stand-by strategy, is QoS-aware, and provides replication as a proactive mechanism.

Bushehrian et al. [24] present an algorithm based on transactional properties for the automatic creation of a compensation workflow given the service dependencies within the composite service. The idea is to minimize the created compensation workflow cost by taking into account the rollback cost of component services. The approach is explained using a case study, but no simulation is done. Compared with our approach, the work of Bushehrian et al. is mostly a design-time approach the runs before composite services are actually executed, since it is focused on the creation of compensation workflows. Also, it differs greatly from our approach regarding the proposed fault tolerance mechanisms.

Lakhal et al. [51] propose a transactional model called FENECIA that includes forward recovery by retying and service replacement, backward recovery by compensation, and the concept of vital and non-vital component services.If a vital component fails and it is not retriable or it has no replacements, the whole composite service execution is aborted. The execution of a composite services may be considered as successful even if non-vital component services failed and were not repaired. To ensure a correct execution order, the composite service execution control is delegated to distributed engines that communicate in a peer-to-peer fashion. Compared with our work, FENECIA proposes the concept of vital and non-vital services, and the AND/OR constructs for the composite service graph, which we do not consider. Our Colored Petri net formal model may be extended

to provide both mechanisms. Among the advantages of our approach over FENECIA we have: the formal modeling of composite services execution processes using Colored Petri nets, the checkpointing mechanism, the replication as a proactive strategy, and the QoS-awareness capabilities.

Liu et al. [55] propose another framework called FACTS to provide fault tolerance for composite services. It is a hybrid fault tolerance approach which combines exception handling and transactional properties. The transactional model and recovery mechanisms are based on FENECIA [51]. The difference between FACTS and FENECIA is that FACTS proposes an implementation for BPEL, while FENECIA proposes a new execution engine. When a failure occurs at runtime, it first employs exception handling strategies to try to repair it. If the failure cannot be fixed, it brings the composite service back to a consistent state using compensation. Fault tolerance rules are specified in a declarative way using Event-Condition-Action rules. Finally, the paper presents an experimental evaluation composed of a case study and a performance evaluation concerning the adoption of the different exception handling strategies. Regarding our work, the same analysis we did for FENECIA applies to FACTS.

6.1.2 Redundancy and Design Diversity-based Approaches

Redundancy and design diversity [52] are common computer science and engineering principles for increasing the reliability of a system. They consist on duplicating the critical components of a system. In the context of fault tolerance for composite services, redundancy and design diversity techniques take the form of replication of component services or service replacement in case of failure. Replication is done by invoking several equivalent services simultaneously, and the response may be taken, for example, from the first one successfully finished, by comparing service responses after all services finished, etc. Hence, replicating a service creates the need for mechanisms to distribute messages, order requests, coordinate replicas, and selecting a response. Replicating services not only increases the availability of the whole composite service but it may also protect against byzantine faults. Some works proposing redundancy and design diversity for composite services are [8, 32, 38, 57, 104].

Dillen et al. [32] propose a classical N-version software fault tolerance approach for redundancy; while Gotze et al. [38] define a majority fault tolerance operator for the enhancement of reliability and availability. Majority is defined as an operation that schedules the same request to all defined services, collects the

results of all successful services, and chooses the most common result. Abdeldjelil et al. [8] describe a voting algorithm for deciding if multiple concurrent service responses are equivalent or not.

Other examples of redundancy and design diversity to provide fault tolerance for composite services are the works of Merideth et al. [57], and Zhou and Wang [104]. Merideth et al. [57] base their approach for byzantine fault tolerance on replica agreement. Zhou and Wang [104] extend Castro and Liskovs Byzantine fault tolerance method [28] to provide byzantine fault tolerance for composite services to protect against arbitrary failures. They focus of their research is to present a protocol which employs a basic replica agreement procedure. Zhou and Wang evaluate their contribution in terms of the overhead generated by replication and replicas agreement. Finally, it is not clear how the approach proposed by Zhou and Wang works for composite services. Since the paper is focused on a replication protocol, it seems it only considers fault tolerance for elementary services instead of composite services.

As a final thought on redundancy and design diversity techniques, we can say that they are useful methods for increasing the availability of elementary services and protecting against byzantine faults. Evidently, increasing the successful execution probability of elementary services has a positive impact on the successful execution probability of composite services; however, redundancy and design diversity techniques do not add either specific or interesting considerations for the fault tolerance design of composite services. Hence, we cannot compare our approach to the presented redundancy and design diversity works, since they are mostly complementary research to this thesis. A byzantine fault tolerance technique may be easily plugged into our framework.

6.1.3 Exception Handling-based Approaches

The strategy described by Simmonds et al. [84] takes into account user guidance to propose several recovery plans. Users manually choose the desired recovery plan among those automatically computed and ranked by the system. It admits the following user guidance: (i) application developers define a set of behavioral correctness properties that need to be maintained at runtime, as well as compensation costs; (ii) application users provide criteria for choosing between possible recovery plans; i.e., based on the plan length, compensation cost, etc; (iii) application users manually choose the desired recovery plan among those automatically

computed, ranked, and proposed by the system. This paper presents a BPEL-based prototype and experimental evaluation using case studies. Compared with our approach, the work of Simmonds et al. is manly about exploiting redundancy to find workarounds when failures occur, and suggest those workarounds to the users. Our approach does not suggest recovery plans, it applies them immediately, as soon as possible; however, our chosen recovery strategies also take into account user guidance in the form of QoS requirements and the composite service execution progress specification.

Sindrilaru et al. [85] propose a fault tolerance approach that extends the ActiveBPEL workflow engine. Their idea is to develop mechanisms for building an autonomic workflow management system that detects, diagnoses, notifies, reacts, and recovers automatically from failures during workflow execution. The detection mechanism inspects the SOAP messages exchanged between the current executing BPEL process and the invoked service. The default behavior of ActiveBPEL may be modified to recover a process from a faulty state, using a non-intrusive checkpointing mechanism. In the checkpointing mechanism, the partial data that might be correct has to be saved and becomes accessible to the user to make any appropriate changes. In general, the work of Sindrilaru et al. focuses on minimizing the time loss in case of an error occurs in the system by detecting failures before they reach the actual workflow engine by intercepting SOAP messages. An experimental evaluation under a test environment is presented to measure the performance of their approach using a case study. In contrast, our work does not depend on SOAP technologies and provides other recovery strategies different to checkpointing. Also, in the work of Sindrilaru et al. it is not clear what happens with the part of the composite service not affected by the detected failure.

Saboohi and Kareem [78] present a two phase approach for composite service fault tolerance. The two proposed phases are: the offline phase, and the online phase. The offline phase refers to an ongoing background process of subgraph computation, while the online phase refers to the moment when a composite service is executing. In the offline phase, the system calculates subgraphs for created composite services. These subgraph are then added to a composite service registry and they can be used later for subgraph replacement. Found subgraphs are ranked according to the semantic description of their component services. The online phase comprises forward recovery through retry and subgraph replacement. If forward recovery does not succeed, backward recovery through compensation is applied. This paper shows experimental evaluation by simulating composite service executions. The goal of the simulation was to illustrate the improvement

of the recovery probability of composite services. In conclusion, the work of Saboohi and Kareem is mainly about the computation of subgraphs during the offline phase, which we consider as a design-time approach. The main different during the offline phase, compared with our work, is that Saboohi and Kareem consider subgraph replacement even if that implies the compensation of some previously successfully executed services. We do not take into account this kind of replacement, but we provide a more complete fault tolerance specification for composite services, and QoS-awareness capabilities which the work of Saboohi and Kareem does not consider.

Brzeziński et al. [22] present an approach called D-ReServE to support service recovery by returning the system to a coherent state. To achieve this, the proposed framework logs interactions between clients and services. These logged interactions are replayed during the recovery procedure. Brzeziński et al. base their model on recovery points which describe a consistent state of the services regardless of fault tolerance mechanisms implemented by service providers. Each service has at least one recovery point available. The only fault tolerance mechanism implemented is roll-back, which is based on the service recovery points. Additionally, messages may be retransmitted periodically to tolerate transient communication failures. Brzeziński et al. evaluate their approach in terms of the overhead introduced by the D-ReServE fault tolerance mechanisms. In contrast, our approach considers a wider range of fault tolerance mechanisms. Also, in our approach compensation is handling automatically using transactional properties; thus, additional mechanisms such as recovery points are not necessary.

Wang et al. [93] propose a framework for the dynamic selection of fault-handling strategies for composite services. The framework contains three components: an exception analyzer, a decision maker, and a strategy selector. The exception analyzer builds a record from the system log of failed services. Next, the decision maker adopts a k-means clustering approach to construct the recovery decision according to fault handling mechanisms of each type of fault. Then, the strategy selector uses an integer program solver to generate an optimal The implemented fault tolerance strategies are skip, service retry, service replacement, and compensation. This paper presents experimental evaluation using case studies to evaluate the effectiveness and performance of the proposed approach. It seems this paper focuses on the optimal selection of the recovery strategy according to its associated QoS. In contrast, our approach takes into account knowledge about the whole composite service execution to deduce recovery and proactive actions using rules.

6.1.4 Prediction and Optimization Approaches

Wenan Tan et al. [89] propose an approach for predicting the performance of composite services. The QoS of component services are evaluated using historical execution data; then, a back propagation neural network model based on particle swarm optimization is used to predict the dynamic performance of the composite service, and to analyze possible QoS violations. Services which violate QoS are detected by computing the correlation between component services and composite service QoS. After these component service are detected, the composite service is optimized by replacing them. This work mostly focuses on the evaluation of QoS time series and prediction. In contrast, our approach does not provide predictions but reactions and adaptations to changes and failures at runtime. Prediction approaches such as the one presented by Wenan Tan et al. may be used to optimize the composite service at design-time in terms of availability, which can be seen as a complement to runtime approaches like ours.

Zheng and Lyu [102] propose an approach for the selection of an optimal fault tolerance strategy for building composite services. In particular, users may provide local and global constraints. Local constraints allow setting QoS maximum values for component services, while global constraints concern the QoS of the whole composite service. The problem of selecting an optimal fault-tolerance strategy is modeled as a 0-1 integer programming problem. A QoS model is introduced to reflect the QoS associate to each fault tolerance strategy; for example, the execution time replication, if the first returned answer is taken, is the minimum time among the estimated execution times of replicas, while the execution time of replicating using N-version programming is the maximum time among the estimated execution times of replicas. Then, starting from an execution plan template, an optimal service selection is done under both local and global constraints. In conclusion, the presented dynamic fault tolerance strategy selection is similar to the service selection problem with additional constraints and considering fault tolerance strategies QoS. In contrast, our approach does not deal with the selection of services. It receives as input the composite service, its corresponding compensation service, equivalent services for component services, etc. Hence, the work of Zheng and Lyu may be considered as a complementary approach to ours.

6.2 Self-healing execution of composite services

Regarding selfhealing approaches, some works build on top of BPEL [19, 59, 61, 86], while others propose new engines [40, 54, 60, 96, 101].

6.2.1 BPEL-based approaches

Modafferi and Conforti [59] present an approach where developers define a BPEL process annotated with recovery information. This BPEL process is then transformed to be executed in a standard BPEL engine. An important part of the paper concerns the proposed abstract extended model, and their corresponding algorithms to transform them into standard BPEL processes. During the design phase, the developer has to design which recovery mechanisms must be used. The supported recovery mechanisms are: modifying process information by external variable setting; task timeout and its corresponding recovery action; re-execution of a service or a part of the composite service; alternative path specification; and rollback and re-execution. No experimental evaluation is presented in this paper. The main differences between the work Modafferi and Conforti and ours is that our fault tolerance approach is automatic following the transactional model, while in the work of Modafferi and Conforti is designed by BPEL developers.

Moser et al. [61] present a system called VieDAME to monitor BPEL processes regarding QoS constraints. It allows the adaptation of existing processes by providing alternative services for a given component service. It also proposed the transformation of SOAP messages to handle service interface mismatches. The implementation is done using Aspect Oriented Programming to intercept SOAP messages and allow services to be replaced at runtime. In general, Moser et al. address two issues of BPEL: it is static by nature, it cannot be changed dynamically at runtime; it does not provide mechanisms for monitoring running processes. The experimental evaluation is done using a case study composed of five services, and it is focused on the running of load tests to compare the performance of VieDAME with the performance of a plain BPEL engine. In contrast, our approach is more about the formal and unambiguous modeling of a self-healing system, instead of particular additions to technologies such as BPEL and SOAP. Additionally, we provide a wider range of fault tolerance mechanisms.

Baresi and Guinea [19] propose Dynamo, which is a rule-based approach to enforce self-healing policies on top of BPEL composite services. It augments the

BPEL technology with supervision rules to set what to check at runtime, and to define how to react when anomalies are found. Each supervision rule states: a location; metadata that influences the evaluation of a rule; monitoring pre- and post-conditions; and a set of reaction strategies. In this paper, the recovery strategies are not explicitly defined, since the focus is on the definition of the supervision rules, and their translation into the language required by the rule engine. Finally, the experimental evaluation concerns the overhead introduced by the monitoring capabilities in comparison with the invocation of plain BPEL activities. This approach is also oriented to BPEL developers, while we provide automatic fault tolerance in our approach. Also, this approach does not provide service replacement as a recovery strategy.

Subramanian et al. [86] propose an extension to BPEL to provide self-healing policies. It allows the definition of pre- and post-conditions of BPEL activities; monitoring; diagnosis; and recovery strategy suggestion. The proposed recovery mechanisms are: service retry, data mediation to solve data and semantic mismatches, service replacement, and subgraph replacement. The BPEL engine is enhanced by introducing a self-healing policy gathering the conditions, monitoring, and recovery mechanisms for BPEL activities. Finally, the paper contains a proof-of-concept prototype illustrating the proposed approach; neither performance analysis nor other evaluations are done in this paper. In contrast to our approach and besides being another extension to BPEL, the approach of Subramanian et al. does not provide replication, compensation or checkpointing mechanisms.

6.2.2 Non-BPEL-based approaches

Halima et al. [40] propose a self-healing framework based on QoS. This framework observes the SOAP messages exchanged between services, and extends them with QoS metadata within their corresponding parameters values. This QoS metadata is used to detect QoS degradation, and react accordingly using service replacement or composite service reconfiguration. The detection of QoS degradation is done by evaluating the general behavior of a given service through time, instead of a specific invocation for that service. The behavior of a service refers to the evolution of its QoS values monitored during its invocations. The experimental evaluation studies the overhead introduced by the QoS monitoring using a case study. In contrast to our approach, the work of Halima et al. is based on the specificities of SOAP, and it only proposes service replacement as recovery

strategy.

Moo-Mena et al. [60] present a self-healing approach for composite services based on QoS degradation. They propose the introduction of a component to intercept exchanged messages between services. The monitoring component receives data collected by the interceptor and sends it to the diagnosis component. Diagnosis takes the data processed by monitoring and analyzes it using the required QoS information. In case of QoS degradation, it warns the recovery module. The recovery module only works if QoS degradation was detected. In case the services performance decreases, recovery will invoke another one to try to meet the performance requirements. If a service fails, recovery replaces it by a new one. Hence, the proposed recovery mechanisms are service retry and replication. In this paper, no experimental evaluation is presented; instead, Moo-Mena et al. place their approach in the context of a real-world application. Similar to the work of Halima et al. [40], this approach is made for SOAP and it only proposes service replacement, which also differs from our approach.

Yin et al. [96] present a self-healing composite service model, which is a combination of compensation and replacement techniques based on QoS and transactional properties. This work focuses on analyzing the cost of compensating composite services. The compensation cost is based on how long after a service was successfully executed its compensation is done; that is, they consider that compensating a service is more expensive as the time passes. An experimental evaluation is done under a simulated environment, and it mainly shows the scalability of the approach. Finally, it is not clear why Yin et al. consider their approach as self-healing research instead of fault tolerance, since they do not mention any of the self-healing principles. Compared with our approach, the work of Yin et al. is QoS-aware regarding only the compensation cost, and it offers limited options concerning fault tolerance strategies.

Li et al. [54] present a self-adaptive approach based on Stochastic Context-free Grammar (SCFG). The authors state that one of the advantages of using SCFG is the possibility of setting fault tolerance strategies as the production rules of the SCFG to choose the optimal service using probabilistic functions. Composite services are represented as automatons where services are connected by transitions. Transition probabilities between services calculated from composite services historical data, and services with higher probabilities will be selected as service replacements. Li and his coauthors take into account service retry and replacement as fault tolerance strategies. For the replacement strategy, local and global decisions are advocated by considering the correlation degree among the

services. In contrast with our work, their approach of is not QoS-aware and it proposes a limited options of fault tolerance strategies.

6.3 Discussion

The work reviewed in this chapter represents existing approaches for the reliable composite service execution, which we have classified as fault tolerance and self-healing research for composite services. The approaches were published between the years 2005 and 2015, and they are the ones we consider as the most relevant to this thesis. Most of the authors evaluate their respective work by doing simulations of composite service executions with the help of case studies. Usually, researchers focus on performance evaluation by measuring the overhead introduced by their proposed approaches and comparing it with plain composite service execution engines with no fault tolerance mechanisms. Due to the absence of an open, close to the real world, and accepted test-bed for composite service executions, and to the complexity and specificities of the implementation of each approach, it is difficult to do a sound comparative analysis quantitatively. We, as well as researchers of reviewed works, mostly limit the comparison among approaches to a qualitative analysis by describing the different approaches in terms of what they offer. To summarize the reviewed works, Table 6.1 shows their publication year, proposed recovery mechanisms, if they are bpel-based or not, if they can be classified as self-healing research, their intrusiveness, and their experimental evaluation method.

Several of the reviewed approaches are propositions to extend the SOAP specification and the WSDL language; for example, by intercepting, analyzing, and modifying exchanged SOAP messages. We have mentioned in Section 2.1 that services based on SOAP have been loosing popularity, while building RESTful services seems to be the preferred option of service developers. This may be seen as a negative point for SOAP-based approaches; nonetheless, their main implemented concepts and techniques are still useful independently of the used technologies. Some of these concepts are: the detection, monitoring, and recovering self-healing loop; redundancy and design diversity techniques for increasing availability and protecting against byzantine faults; roll-back and compensation; data mediation to solve data and semantic mismatches; service and subgraph replacement; checkpointing; and prediction and optimization techniques. In addition, many fault tolerance approaches are supported by exception handling constructs at the language level. We believe that fault tolerance must be handled

at a higher level of abstraction. Finally, some reviewed works are a mix between design-time and runtime techniques. While we do not deny the importance of design-time techniques, in this thesis we are interested on what happens and the decision making process at runtime.

In this thesis, first of all we aim to formalize composite services, their execution processes, fault tolerance mechanisms, and self-healing capabilities unambiguously. This formal model is language and technology independent. Then, our vision is to have smart software agents capable of representing a component service during a composite service execution. The idea is to have only one type of agent, called Service Agent, responsible for the actual execution of a service independently of the technology in which that service was developed. Services Agents will be capable of making decisions by being context- and self-aware. Among the fault tolerance strategies our approach provides we have: backward recovery by compensation, forward recovery by service retrying and replacement, replication, and checkpointing. Finally, our vision is to study services at a conceptual level to propose concepts and techniques that may be easily implemented regardless of the underlying technologies; for example, the proposed Service Agents may be the abstractions of SOAP or RESTful services, and those services may have physical counterparts in the real world or they may be the exposed functionalities of software systems.

		Recovery mechanism											Eval.	
approach	*publication year*	*transactional properties*	*compensation*	*retry*	*substitution*	*replication*	*checkpointing*	*replaning*	*ignore*	*ws-bpel*	*self-healing*	*intrusive*	*simulation*	*case study*
Wenan Tan et al. [89]	2015	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓
Zheng et al. [102]	2015	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓
Wang et al. [93]	2015	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓
Li et al. [54]	2014	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✓	✓
Bushehrian et al. [24]	2012	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Saboohi and Abdul [78]	2012	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
Behl et al. [20]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓	✓
Brzeziński and et al. [22]	2012	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Dillen et al. [32]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Abdeldjelil et al. [8]	2012	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Cardinale and Rukoz [25]	2011	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Liu et al. [55]	2010	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓
Simmonds et al.[84]	2010	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓
Sindrilaru et al. [85]	2010	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓
Zhou and Wang [104]	2010	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Lakhal et al. [51]	2009	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗
Yin et al. [96]	2009	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Moser et al. [61]	2008	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗	✓	✓
Subramanian et al. [86]	2008	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓
Halima et al. [40]	2008	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓
Moo-Mena et al. [60]	2008	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗
Gotze et al. [38]	2008	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Baresi and Guinea [19]	2007	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓
Modafferi and Conforti [59]	2006	✗	✓	✓	✗	✗	✗	✓	✗	✓	✓	✗	✗	✗
Merideth et al. [57]	2005	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Our approach	**2015**	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓

Table 6.1: Reviewed Work By Publication Year.

Chapter 7

General Conclusions

Contents

7.1 Summary **153**

7.2 Limitations **155**

7.3 Future Research Directions **155**

7.3.1 Fault Identification and Reaction 156

7.3.2 Self-healing Internet of Things Applications 156

7.3.3 Composite Service Execution and Big Data 157

7.1 Summary

In this thesis, we have proposed a self-healing composite service approach based on transactional properties and knowledge-based agents. Our approach is situated in the Service Composition & Execution layer of the Service Oriented Architecture middleware presented in Chapter 2. One of the main advantages of our work is the use of transactional properties as a deep-seated notion for fault tolerance. This way, we implemented backward and recovery mechanisms which function automatically without the need of developers or any other kind of human intervention. Then, we extended our transactional approach with knowledge-based agents capable of representing facts about a composite service execution, and deducing new information from those facts. The new deduced information plays a crucial role in the decision making process at runtime. Another important consideration is the fact that our work deals with services at a conceptual level

to provide a formal model for composite service executions. This model is composed by concepts and techniques that may be easily implemented regardless of the underlying technologies. Actual services may be SOAP or RESTful services, and they may have physical counterparts in the real world, or they may be the exposed functionalities of software systems or resources. Last but not least, our work also sets the basis for interesting future research such as the managing and self-healing aspects of distributed applications in the Future Internet. We propose a summary of the chapters of this thesis in the following paragraphs.

In Chapter 3, we formalized composite services and their fault tolerance execution processes using Colored Petri nets. We then proposed a framework composed by two type of components: an Agent Coordinator responsible of managing the global aspects of composite service executions; and, Service Agents which execute the actual services and are in charge of the fault tolerance execution control. Our framework ensures the correct and fault tolerant execution of composite services, and its distributed execution model may be implemented in distributed or shared memory systems. The provided fault tolerance mechanisms are backward recovery by compensation, forward recovery by retry and service replacement, and checkpointing as an alternative stand-by strategy.

In Chapter 4, we have introduced a self-healing composite service approach that extends the transactional approach of Chapter 4. We formulated a hypothesis highlighting the need of providing dynamism regarding fault tolerance strategy selection and QoS monitoring. In our self-healing approach, Service Agents become knowledge-based agents. They make recovery and proactive fault tolerance strategy selection based on the information they have about the whole composite service, about themselves, and about what is expected and what it is really happening at runtime.

In Chapter 5, we presented an implementation of our approach and evaluated it experimentally using a case study. We implemented three different systems: a composite service execution system with no fault tolerance, the transactional approach of Chapter 3, and a self-healing approach of Chapter 4. The experimental evaluation showed three main observation: (i) the importance of providing fault tolerance mechanisms for composite services by analyzing the results of the system with no fault tolerance; (ii) how the approach of Chapter 3 handles composite service failures using transactional properties regardless of QoS degradation; and, (iii) how the self-healing approach of Chapter 4 builds on top of our transactional approach to provide more sophisticated decision making through self- and context-awareness. Additionally, experimental evaluation also showed that both

transactional and self-healing approaches may be implemented without adding significant overhead to the composite service execution control. The evaluation presented in this chapter suggests that combining transactional properties with self-healing capabilities leads to smarter execution systems with the ability to handle higher level goals for composite service executions with minimal human intervention.

In Chapter 6, we proposed a review of existing approaches to support the reliable execution of composite services. This review includes approaches that may be classified as fault tolerance and self-healing research. We selected the most relevant works between the years 2005 and 2015, and we compared them with the approach proposed in this thesis. We highlighted the importance of having technology-independent smart agents capable of representing a component service during a composite service execution. These smart agents should possess knowledge about the application they participate in, and be able of making decisions by reasoning about this knowledge.

7.2 Limitations

The main limitations of this thesis are the following:

1) the lack of a testbed accepted by the research community to do sound experimental evaluation. As we saw in Chapter 6, researchers usually test their approaches by simulating study cases.

2) the difficulty to deploy our approach in the real world due to the lack of automation and interoperability between services published in the Internet.

Note that these exposed limitations concern all the research on composite service execution, and not only this thesis.

7.3 Future Research Directions

We envisage interesting possibilities by taking this thesis as point of departure. Our future research concerns the implementation of more sophisticated fault identification and reaction mechanisms, the definition of a self-healing framework for

the Internet of Things, and the analysis of the data generated by our composite service execution system.

7.3.1 Fault Identification and Reaction

In Section 2.2.1, we presented the fault hypothesis considered in this thesis. We talked bout the types of faults from which component services can suffer from, and which ones our fault tolerance composite service approach handles. In our approach, we do not identify the type of faults; instead, we handle general service failures which may be caused by any of the considered faults. In case of failure, we applied recovery mechanisms according to the availability of those mechanisms and user preferences. Nonetheless, it is important to identify the type of faults instead of general service failures since different faults may require different reactions [29]. For example, a time-out fault may be solved by retrying the failed service, while other faults may require service replacement.

Also in Section 2.2.1 we have stated that services are managed by reliable component which do not fail. Later on, in Chapter 3, we have called these components Service Agent. In Chapter 4, we haven seen that the development of these Service Agent is heading towards autonomous components which are smarter abstraction of services. Hence, it may be pertinent to consider Service Agent failures; for example, a Service Agent participating in a composite service execution may not respond.

7.3.2 Self-healing Internet of Things Applications

Researchers recognize the importance of the automatic execution and fault tolerance of services as a critical feature in the Internet of Things paradigm [68]. As we have explained at the beginning of this thesis, in Section 2.1, we showed that services no longer are only software operations exposed on the Internet, but also the abstraction of physical objects capable of modifying the physical world. It is crucial to recognize this emerging nature of service to face the new challenges introduced by composite services which interact with both virtual and physical worlds. Failures in this type of composite services may lead to loss of production time, equipment damage, environmental catastrophes, or loss of human life [9].

Mrissa et al. [63] introduce the concept of avatar as a virtual extension to objects. These avatars will exhibit autonomous behavior and collaboration, and

can be deployed directly on objects, or on a cloud infrastructure for resource-constrained objects. The concept of Service Agent is not far from the one of avatars; therefore, we plan to extend and adapt Service Agents to exhibit avatar-like capabilities. For example, instead of instantiate a new Service Agent responsible for a component service each time a composite service execution starts, Service Agent will join composite applications to collaborate in the achievement of the desired goal. Also, we will have an Application Coordinator instead of an Agent Coordinator. The idea is to have an Application Coordinator per critical application. A critical application is a composite service with high availability and fault tolerance requirements such as e-health [23], Industry 4.0 [68], or any other of the applications presented in [9]. The Application Coordinator will manage the participating Service Agents, given them the required information to achieve high-level goals, and trigger emergency mechanisms if needed. Also, it will show the application health, other relevant information, and provide administration facilities through a dashboard accessible to human users. For achieving integration between objects and Service Agents, it is crucial to understand the nature and capabilities of objects since handling the heterogeneity of Internet of Things applications still a challenge [69]. A lightweight RESTful approach may be taken for building our envisaged Internet of Things framework [39].

It is also important to define the meaning of transactionality in the context of the Internet of Things applications. Concepts like rollback, compensation, replacement, and replication may have special considerations due to the fact that we are dealing with services with the capability of changing the physical world.

7.3.3 Composite Service Execution and Big Data

The number of services published in the Internet has been growing since their introduction; moreover; the introduction of connected objects has made this number of services increase even faster [91]. Internet of Things predictions state that there will be more than 16 billion connected objects by the year 2020 [87]. As we have seen in Section 2.1, these connected objects will expose their functionalities as services.

One of the consequences of this explosion of connected objects if the generation huge quantities of data; therefore, it is expected that related messaging volumes could easily reach between 1000 and 10000 per person per day [91]. Researchers acknowledge that one of the most important challenges is the handling and anal-

ysis of all the data generated by these connected objects, since it will only be of value if it is collected, analyzed, and interpreted [75, 53, 90].

In our context, the vision is to collect and store the data generated by our composite service execution system, including component service behavior, selected strategies and their impact on the composite service execution. This data may be analyzed to improve the selection of replacement services and the decision making of recovery and proactive strategies.

Appendix A

Algorithms

In this chapter, we review some algorithms used throughout this thesis. In particular, we show the Critical Path method used to compute knowledge about estimated execution times (Def. 4.3.5, 4.3.8, and 4.3.9). The Critical Path method we implement is a depth-first search based algorithm. We also present the algorithm to compute the predecessors (Def. 4.3.15) and dependent outputs (Def. 4.3.14) for a given service.

A.1 Expected Execution Time Knowledge and The Critical Path Method

We implemented the Topological Sort algorithm to compute the Critical Path of a composite service. Further information about the basic algorithm can be found in the Chapter 22 called *Elementary Graph Algorithms* of the book *Introduction to Algorithms, Third Edition* of Cormen et al. [31].

The first step is to use the Depth-first search algorithm (page 604 of [31]) to compute a list of services in topological order (page 613 of [31]). Once we have computed the topological order of the composite service graph, we can find the minimum time to execute all services. Essentially, we have to find the longest path in the composite service graph, since the minimum amount of time needed to execute all services is the time needed to execute the chain of services with the longest execution time. Algorithm A.1 shows how to compute the longest path given the topological sort of a composite service graph and a source service. The

result is a list of services with their corresponding minimum time to finish their executions.

Algorithm A.1 Longest Path

```
Input: Topological Sort ts, source service source
Output: Distances d
begin
    // Initialize distances to all services as infinite and distance
    // to source as 0
    ∀s ∈ S, d[s] ← ∞
    d[source] ← 0
    for s ∈ ts do
        // Update distances of all adjacent services
        if d[s] ≠ ∞ then
            for succ ∈ (s•)• do
                if d[s] + s≈time > d[succ] then
                    d[succ] ← d[s] + s≈time
                end
            end
        end
    end
end
```

A.1.1 Critical Path Example

Let us go back to our case study depicted in Figure 5.7 on page 122 Section 5.2.2. If we run the topological sort algorithm with the graph representing this composite service as input, we obtain the topological order showed in Figure A.1.

Now, we can compute the critical path from any service; for example, Table A.1 shows the output of Algorithm A.1 for the initial service. Column $d[]$ shows the minimum completion time for each service from the initial service; we can see that

$$\begin{aligned} d[FinalService] &= 108286.10999999999 \\ &= \circ^{\approx}_{time} \end{aligned}$$

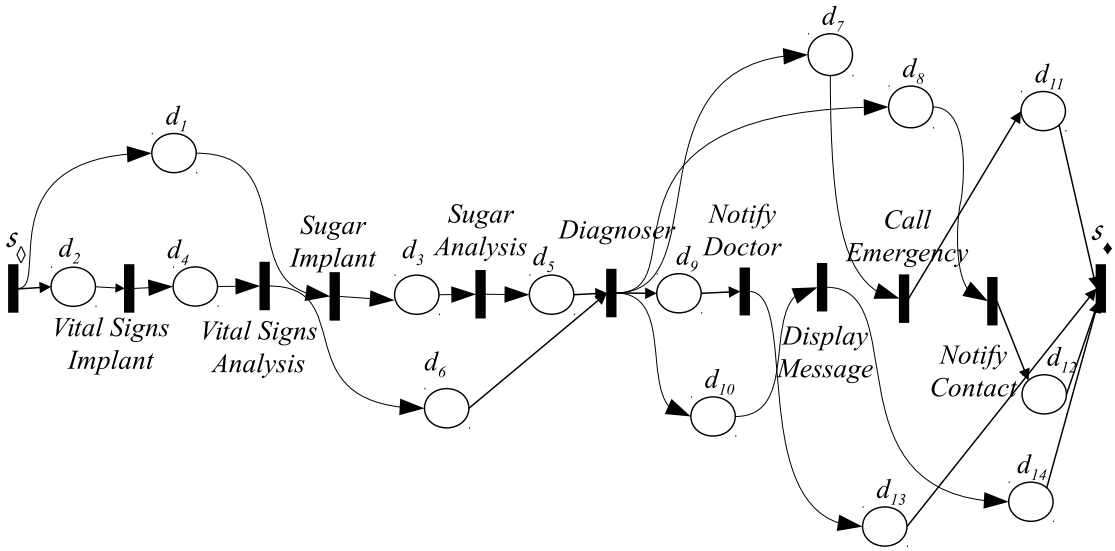


Figure A.1: Visual Representation of the Topological Sorting of the e-Health Study Case of Figure 5.7

which satisfies Def. 4.3.5, and is actually the estimated execution time $\circ^{\approx}_{time}$ of the whole composite service. We can also know the firing time (Def. 4.3.8) of each service by looking at their corresponding values in column $d[]$ of Table A.1. Let us take the *Diagnoser* service as example. If we look at its value in Table A.1, we see that:

$$\begin{aligned} d[Diagnoser] &= 74057.23 \\ &= Diagnoser^{\approx}_{firing} + Diagnoser^{\approx}_{time} \end{aligned}$$

therefore

$$\begin{aligned} Diagnoser^{\approx}_{firing} &= d[Diagnoser] - Diagnoser^{\approx}_{time} \\ &= 74057.23 - 11671.86 \\ &= 62385.37 \end{aligned}$$

Finally, Table A.2 shows the result of Algorithm A.1 for every service in the component service. From this table, we can obtain the remaining time (Def. 4.3.9) for each component service. The column on the left of the table shows all the

services in the component service; the rest of the columns show the computed values for each service, where $-\infty$ means that the service on the left is not reachable from the current service. For example, if we look at the computed valued for the *Diagnoser* service in Table A.2, we have that:

$$\begin{aligned} d[\gamma_{\Diamond}, Diagnoser] &= d[VitalSignsAnalysis, Diagnoser] \\ &= d[VitalSignsImplant, Diagnoser] \\ &= d[SugarAnalysis, Diagnoser] \\ &= d[SugarImplant, Diagnoser] \\ &= -\infty \end{aligned}$$

since none of the $\gamma_{\Diamond}$, $VitalSignsAnalysis$, $VitalSignsImplant$, $SugarAnalysis$, $SugarImplant$ services are reachable from *Diagnoser*. To know the remaining time for *Diagnoser*, it suffices to look at the computed value for $\gamma_{\blacklozenge}$, as follows:

$$\begin{aligned} d[\gamma_{\blacklozenge}, Diagnoser] &= 34228.88 \\ &= Diagnoser^{\approx}_{remaining} \end{aligned} \tag{A.1}$$

Note that Eq. A.1 satisfies Def. 4.3.9 since:

$$\begin{aligned} Diagnoser^{\approx}_{remaining} = max(&NotifyDoctor^{\approx}_{time} + NotifyDoctor^{\approx}_{remaining}, \\ &CallEmergency^{\approx}_{time} + CallEmergency^{\approx}_{remaining}, \\ &NotifyContact^{\approx}_{time} + NotifyContact^{\approx}_{remaining}, \\ &DisplayMessage^{\approx}_{time} + DisplayMessage^{\approx}_{remaining}) \\ &= NotifyDoctor^{\approx}_{time} + NotifyDoctor^{\approx}_{remaining} \\ &= NotifyDoctor^{\approx}_{time} + 0.0 \\ &= 34228.88 \end{aligned}$$

This time represents the remaining time from *Diagnoser* until the end of the composite service execution. Now, we can calculated the local global time (Def. 4.3.10) of *Diagnoser* as follows:

	$d[]$
$\gamma_\lozenge$	0.0
VitalSignsAnalysis	62385.369999999995
NotifyDoctor	108286.10999999999
DisplayMessage	95377.95999999999
CallEmergency	90180.89
VitalSignsImplant	51712.56
Diagnoser	74057.23
SugarAnalysis	39667.1
$\gamma_\blacklozenge$	108286.10999999999
SugarImplant	12263.33
NotifyContact	75973.37

Table A.1: Critical Path from $\gamma_\lozenge$ to $\gamma_\blacklozenge$

$$\begin{aligned} Diagnoser(\circ)^{\approx}_{time} &= Diagnoser^{\approx}_{firing} + Diagnoser^{\approx}_{time} + Diagnoser^{\approx}_{remaining} \\ &= 62385.37 + 11671.86 + 34228.88 \\ &= 108286.10999999999 \end{aligned}$$

In this case, $Diagnoser(\circ)^{\approx}_{time} = \circ^{\approx}_{time}$ since *Diagnoser* is in the critical path of the composite service.

Finally, by looking at Tables A.1 and A.2, we may follow the same procedure for every service in the composite service to compute their execution time-related knowledge.

A.2 Predecessors and Dependent Outputs

Computing the predecessors and dependent outputs of a given service can be seen as a reachability problem. Given a source service, all services we can reach traversing the composite service graph starting from its adjacent predecessors count as predecessors of the source service. In the same way, the outputs of all composite service outputs we can reach traversing the composite service graph are dependent outputs of the source service.

	VitalSignsAnalysis	*NotifyDoctor*	*DisplayMessage*	*CallEmergency*	*VitalSignsImplant*	*Diagnoser*	*SugarAnalysis*	*SugarImplant*	*NotifyContact*
$\gamma_\diamond$	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
VitalSignsAnalysis	0.0	-∞	-∞	-∞	10672.81	-∞	-∞	-∞	-∞
NotifyDoctor	45900.74	0.0	-∞	-∞	56573.549999999996	34228.88	45900.74	73304.51000000001	-∞
DisplayMessage	32992.59	-∞	0.0	-∞	43665.399999999994	21320.73	32992.59	60396.36	-∞
CallEmergency	27795.52	-∞	-∞	0.0	38468.33	16123.66	27795.52	55199.29000000001	-∞
VitalSignsImplant	-∞	-∞	-∞	-∞	0.0	-∞	-∞	-∞	-∞
Diagnoser	11671.86	-∞	-∞	-∞	22344.67	0.0	11671.86	39075.630000000005	-∞
SugarAnalysis	-∞	-∞	-∞	-∞	-∞	-∞	0.0	27403.7	-∞
$\gamma_\blacklozenge$	45900.74	0.0	0.0	0.0	56573.549999999996	34228.88	45900.74	73304.51000000001	0.0
SugarImplant	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
NotifyContact	13588.0	-∞	-∞	-∞	24260.809999999998	1916.14	13588.0	40991.770000000004	0.0

Table A.2: Critical Path from each component service to $\gamma_\blacklozenge$

In our approach, the Agent Coordinator can perform the required computation, in exchange for preprocessing time and some extra storage, using a basic graph algorithm such as Bread-first search. Again, we implemented a basic Bread-first search algorithm, which is explained on page 595, Chapter 22, of *Introduction to Algorithms, Third Edition* [31].

Appendix B

Experiences on Random Composite Services

In this chapter, we present the execution time of the main algorithms used in this thesis. This analysis is done on random composite services generated using the Barabási-Albert model [18], which is an algorithm for random generation of scale-free networks using a preferential attachment mechanism. We used the random graph generator provided by the JUNG library [1]. All algorithms were ran in a PC with the following configuration: Intel Core i5-3210M CPU @ 2.50GHz ×4; 1GB RAM, memory of 3.8GiB; Ubuntu 14.04 LTS 32-bit, and Java 7.

The rest of this chapter is structured as follows: in Section B.1 we present the estimated execution times our randomly generated composite services, the execution time of the Critical Path algorithm (Section A.1) to compute their estimated execution times, and the overhead of the Critical Path algorithm on the composite services execution times; in Section B.2 we show the estimated price and availability of our random composite services; and in Section B.3 we show the execution times of algorithms to compute the dependent outputs and progress of our random composite services.

[1] http://jung.sourceforge.net/

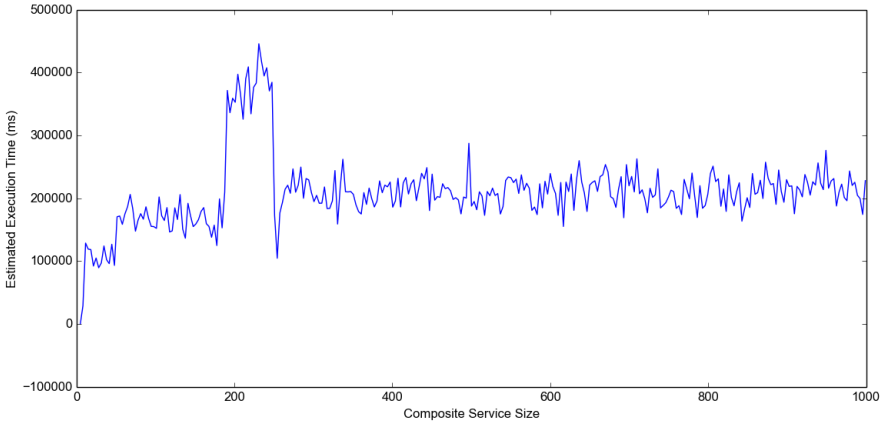


Figure B.1: Random Composite Services Estimated Execution Times.

B.1 Estimated Execution Time and the Critical Path Algorithm

Figure B.1 shows the estimated execution times of our random composite services. Given the nature of the generated composite services graphs, the estimated execution time does not change much. This is due to the fact the new services are added in parallel to other services and their estimated execution time does not affect the Critical Path of the composition.

Figure B.2 shows the execution times of the Critical Path algorithm for calculating the estimated execution times of our random composite services.

Figure B.3 shows the overhead of Critical Path algorithm. We compute this overhead as the percentage of the estimated execution time (Figure B.1) taken to compute the Critical Path.

B.2 Estimated Price and Availability

Figure B.4 shows the global price and Figure B.5 shows the global availability of our random composite services. We do not show the execution time of computing the global prices and availabilities; both computations run in $\mathcal{O}(n)$ and they are close to zero for all composite services. Note that composite services availabilities

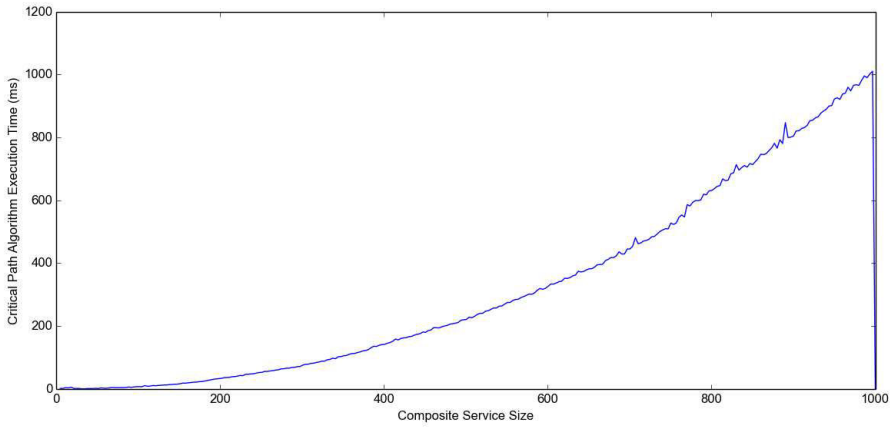


Figure B.2: Critical Path Algorithm Execution Times on Random Composite Services.

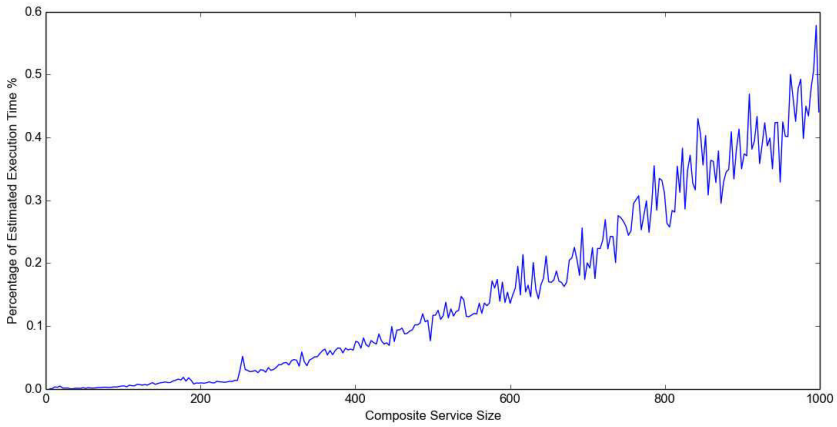


Figure B.3: Critical Path Algorithm Impact on Random Composite Services Estimated Execution Times.

are low and decrease quickly, and for compositions of more than 40 services, the availability is 0.

B.3 Dependent Outputs and Predecessors

Figure B.6 shows the execution time taken to compute the dependent outputs (Section A.2) of component services of each of our random composite services.

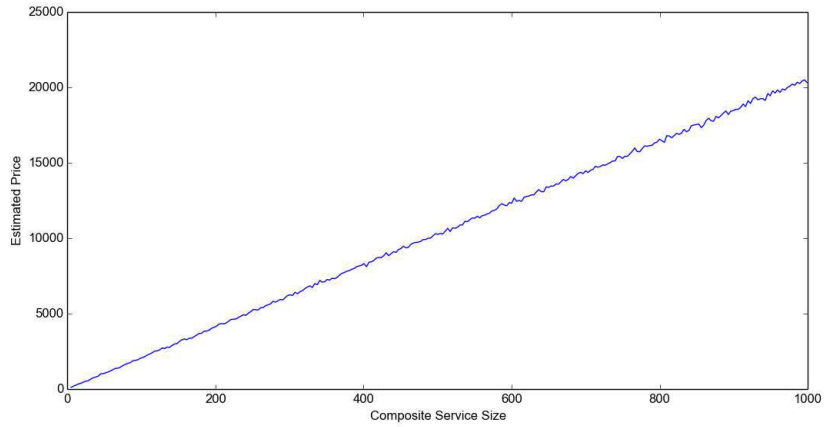


Figure B.4: Random Composite Services Prices.

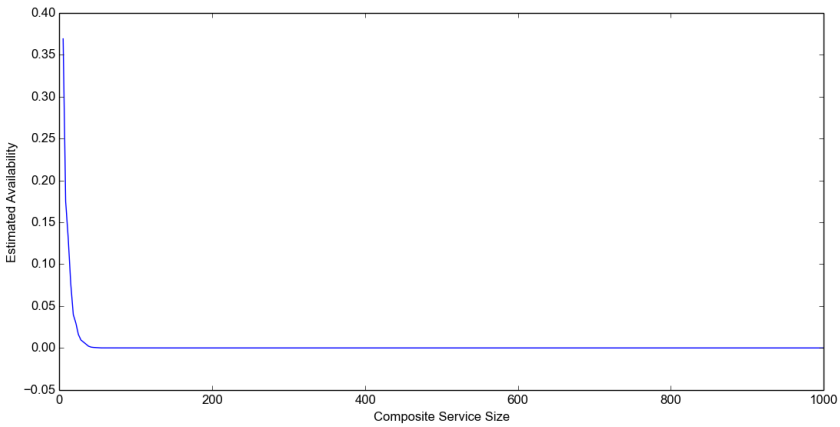


Figure B.5: Random Composite Services Availabilities.

Figure B.7 shows the execution time taken to compute the predecessors (Section A.2) of component services of each of our random composite services.

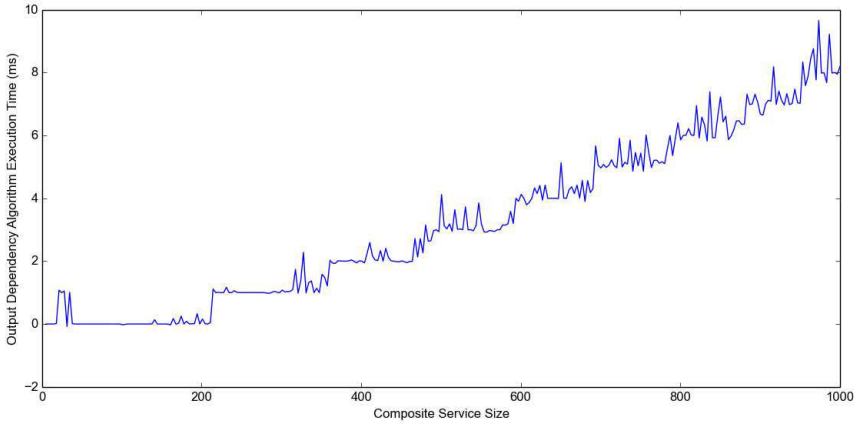


Figure B.6: Dependent Output Algorithm Execution Times on Random Composite Services.

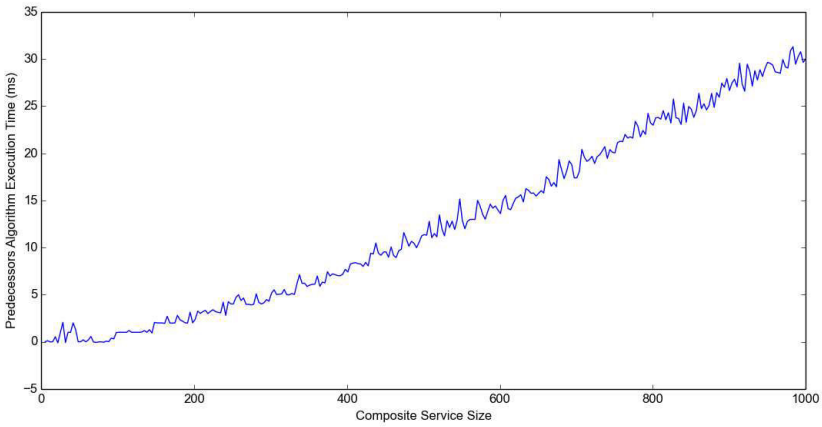


Figure B.7: Predecessors Algorithm Execution Times on Random Composite Services.

Bibliography

[1] National Intelligence Council, Disruptive Civil Technologies Six Technologies with Potential Impacts on US Interests Out to 2025 Conference Report CR 2008-07, April 2008. http://www.dni.gov/nic/NIC_home.html.

[2] OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed: June 2015.

[3] OWL 2 Web Ontology Language Document Overview (Second Edition). http://www.w3.org/TR/owl2-overview/. Accessed: June 2015.

[4] The Open Group. The SOA Work Group. http://www.opengroup.org/soa/. Accessed: June 2015.

[5] OASIS. Business transaction protocol Version 1.0. http://docs.oasis-open.org/ws-tx/wsba/2006/06, 2002. Accessed: November 2015.

[6] OASIS. Web Services Composite Application Framework Version 1.0. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf, 2005. Accessed: November 2015.

[7] OASIS. Web Services Business Activity Version 1.2. https://www.oasis-open.org/committees/business-transaction/documents/primer/Primerhtml/BTP%20Primer%20D1%2020020602.html, 2009. Accessed: November 2015.

[8] H. Abdeldjelil, N. Faci, Z. Maamar, and D. Benslimane. A Diversity-Based Approach for Managing Faults in Web Services. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 81–88, March 2012.

[9] Pekka Alho and Jouni Mattila. Service-oriented Approach to Fault Tolerance in CPSs. *J. Syst. Softw.*, 105(C):1–17, July 2015.

[10] Rafael Angarita. Dynamic Composite Web Service Execution by Providing Fault-Tolerance and QoS Monitoring. In *Service-Oriented Computing - ICSOC 2014 Workshops and Satellite Events, Paris, France, November 3-6, 2014, Revised Selected Papers*, pages 371–377, 2014.

[11] Rafael Angarita. Responsible Objects: Towards Self-Healing Internet of Things Applications. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 307–312, July 2015.

[12] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. FaCETa: Backward and Forward Recovery for Execution of Transactional Composite WS. In *Proceedings of the Fifth International Workshop on REsource Discovery (RED 2012)*, pages 1–15, Heraklion, Grece, 2012.

[13] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Dynamic Recovery Decision During Composite Web Services Execution. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, MEDES '13, pages 187–194, New York, NY, USA, 2013. ACM.

[14] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Reliable Composite Web Services Execution: Towards a Dynamic Recovery Decision . *Electronic Notes in Theoretical Computer Science*, 302(0):5 – 28, 2014.

[15] Rafael Angarita, Maude Manouvrier, and Marta Rukoz. A Framework for Transactional Service Selection Based on Crowdsourcing. In *Mobile Web and Intelligent Information Systems*, volume 9228 of *Lecture Notes in Computer Science*, pages 137–148. Springer International Publishing, 2015.

[16] Rafael Angarita, Marta Rukoz, and Yudith Cardinale. Modeling dynamic recovery strategy for composite web services execution. *World Wide Web*, pages 1–21, 2015.

[17] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[18] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[19] L. Baresi and S. Guinea. Dynamo and Self-Healing BPEL Compositions. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 69–70, May 2007.

[20] J. Behl, T. Distler, F. Heisig, R. Kapitza, and M. Schunter. Providing Fault-tolerant Execution of Web-service based Workflows within Clouds. In *Proc. of the 2nd Internat. Workshop on Cloud Computing Platforms (CloudCP)*, 2012.

[21] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[22] Jerzy Brzezinski, Arkadiusz Danilecki, Mateusz Holenko, Anna Kobusinska, Jacek Kobusinski, and Piotr Zierhoffer. D-ReServE: Distributed Reliable Service Environment. In *ADBIS*, volume 7503 of *LNCS*, pages 71–84, 2012.

[23] Nicola Bui and Michele Zorzi. Health Care Applications: A Solution Based on the Internet of Things. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, ISABEL '11, pages 131:1–131:5, New York, NY, USA, 2011. ACM.

[24] O. Bushehrian, S. Zare, and N. Keihani Rad. A Workflow-Based Failure Recovery in Web Services Composition. *Journal of Software Engineering and Applications*, 5:89–95, 2012.

[25] Yudith Cardinale and Marta Rukoz. A Framework for Reliable Execution of Transactional Composite Web Services. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '11, pages 129–136, New York, NY, USA, 2011. ACM.

[26] Yudith Cardinale, Marta Rukoz, and Rafael Angarita. Modeling Snapshot of Composite WS Execution by Colored Petri Nets. In *Resource Discovery*, volume 8194 of *Lecture Notes in Computer Science*, pages 23–44. Springer Berlin Heidelberg, 2013.

[27] Rubn Casado, Muhammad Younas, and Javier Tuya. A Generic Framework for Testing the Web Services Transactions. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 29–49. Springer New York, 2014.

[28] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[29] K.S.May Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A Fault Taxonomy for Web Service Composition. In *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 363–375. Springer Berlin Heidelberg, 2009.

[30] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu, and Hucheng Wang. A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective. *Internet of Things Journal, IEEE*, 1(4):349–359, Aug 2014.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[32] Roeland Dillen, Jonas Buys, Vincenzo De Florio, and Chris Blondia. WSDM-Enabled Autonomic Augmentation of Classical Multi-version Software Fault-Tolerance Mechanisms. In *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 294–306. Springer Berlin Heidelberg, 2012.

[33] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[34] Joyce El Haddad, Maude Manouvrier, and Marta Rukoz. TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition. *IEEE Trans. Serv. Comput.*, 3(1):73–85, January 2010.

[35] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[36] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987.

[37] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing Systems - Survey and Synthesis. *Decis. Support Syst.*, 42(4):2164–2185, January 2007.

[38] J. Gotze, J. Muller, and P. Muller. Iterative Service Orchestration based on Dependability Attributes. In *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, pages 353–360, Sept 2008.

[39] Tor-Morten Grnli, Gheorghita Ghinea, and Muhammad Younas. A Lightweight Architecture for the Web-of-Things. In *Mobile Web Information Systems*, volume 8093 of *Lecture Notes in Computer Science*, pages 248–259. Springer Berlin Heidelberg, 2013.

[40] Riadh Ben Halima, Khalil Drira, and Mohamed Jmaiel. A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services. In *Proceedings of the 2008 IEEE International Conference on Web Services*, ICWS '08, pages 104–111, Washington, DC, USA, 2008. IEEE Computer Society.

[41] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *Proceedings of the 14th Australasian Database Conference - Volume 17*, ADC '03, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[42] Son N. Han, Imran Khan, Gyu Myoung Lee, Noel Crespi, and Roch H. Glitho. Service composition for IP smart object using realtime Web protocols: Concept and research challenges. *Computer Standards & Interfaces*, 43:79 – 90, 2016.

[43] M.N. Huhns and M.P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, Jan 2005.

[44] IBM. Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM, 2001.

[45] IBM. An architectural blueprint for autonomic computing. IBM, 2005.

[46] ISO. Automation systems and integration Product data representation and exchange. ISO 10303, International Organization for Standardization, 2004.

[47] ISO. Quality management systems. ISO 9000:2005, International Organization for Standardization, 2009.

[48] Kurt Jensen. Coloured Petri nets. In *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.

[49] James E Kelley Jr. Critical-path planning and scheduling: Mathematical basis. *Operations research*, 9(3):296–320, 1961.

[50] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[51] Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. FENECIA: failure endurable nested-transaction based execution of compo site Web services with incorporated state analysis. *VLDB Journal*, 18(1):1–56, 2009.

[52] Peter A Lee and Thomas Anderson. *Fault tolerance: principles and practice*, volume 3. Springer Science & Business Media, 2012.

[53] Martin Lehmann, Andreas Birn-Hansen, Gheorghita Ghinea, Tor-Morten Grønli, and Muhammad Younas. Data Analysis as a Service: An Infrastructure for Storing and Analyzing the Internet of Things. In *Mobile Web and Intelligent Information Systems*, volume 9228 of *Lecture Notes in Computer Science*, pages 161–169. Springer International Publishing, 2015.

[54] Guoqiang Li, Lejian Liao, Dandan Song, and Zhenling Zhang. Self-Adaptive Web Service Composition Based on Stochastic Context-Free Grammar. In *e-Business Engineering (ICEBE), 2014 IEEE 11th International Conference on*, pages 139–144, Nov 2014.

[55] An Liu, Qing Li, Liusheng Huang, and Mingjun Xiao. FACTS: A Framework for Fault-Tolerant Composition of Transactional Web Services. *Services Computing, IEEE Transactions on*, 3(1):46–59, Jan 2010.

[56] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 242–259. Springer Berlin Heidelberg, 2010.

[57] M.G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for Web-service applications. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 131–140, Oct 2005.

[58] Elena Meshkova, Janne Riihijrvi, Marina Petrova, and Petri Mhnen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks . *Computer Networks*, 52(11):2097 – 2128, 2008.

[59] Stefano Modafferi and Eugenio Conforti. Methods for Enabling Recovery Actions in Ws-BPEL. In *Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I*, ODBASE'06/OTM'06, pages 219–236, Berlin, Heidelberg, 2006. Springer-Verlag.

[60] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, and F. Alonzo-Canul. Defining a Self-Healing QoS-based Infrastructure for Web Services Applications. In *Computational Science and Engineering Workshops, 2008. CSEWORKSHOPS '08. 11th IEEE International Conference on*, pages 215–220, July 2008.

[61] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 815–824, New York, NY, USA, 2008. ACM.

[62] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.

[63] M. Mrissa, L. Medini, J.-P. Jamont, N. Le Sommer, and J. Laplace. An Avatar Architecture for the Web of Things. *Internet Computing, IEEE*, 19(2):30–38, Mar 2015.

[64] Amanda S. Nascimento, Cecília M. F. Rubira, Rachel Burrows, and Fernando Castor. A Systematic Review of Design Diversity-based Solutions for Fault-tolerant SOAs. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, pages 107–118, New York, NY, USA, 2013. ACM.

[65] OASIS. Web Services Businnes Process Execution Language (WS-BPEL), Version 2.0. http://docs.oasis-open.org/wsbepel/2.0/wsbpel-v2.0.html, 2007. OASIS Standard.

[66] Massimo Paolucci, Takahiro Kawamura, TerryR. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In Ian Horrocks and James Hendler, editors, *The Semantic Web ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer Berlin Heidelberg, 2002.

[67] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[68] C. Perera, C.H. Liu, S. Jayawardena, and Min Chen. A Survey on Internet of Things From Industrial Market Perspective. *Access, IEEE*, 2:1660–1679, 2014.

[69] M.E. Perez Hernandez and S. Reiff-Marganiec. Classifying Smart Objects using capabilities. In *Smart Computing (SMARTCOMP), 2014 International Conference on*, pages 309–316, Nov 2014.

[70] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[71] Harald Psaier and Schahram Dustdar. A Survey on Self-healing Systems: Approaches and Systems. *Computing*, 91(1):43–73, January 2011.

[72] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 26–37, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[73] Shuping Ran. A Model for Web Services Discovery with QoS. *SIGecom Exch.*, 4(1):1–10, March 2003.

[74] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2005.

[75] Mark Raskino, Jackie Fenn, and Alexander Linden. Extracting value from the massively connected world of 2015. *Gartner Res., Stamford, CT, USA, Tech. Rep. G*, 125949, 2005.

[76] Leonard Richardson and Sam Ruby. *Restful Web Services*. O'Reilly, first edition, 2007.

[77] Marta Rukoz, Yudith Cardinale, and Rafael Angarita. FACETA*: Checkpointing for Transactional Composite Web Service Execution based on Petri-Nets . *Procedia Computer Science*, 10(0):874 – 879, 2012.

[78] Hadi Saboohi and Sameem Abdul Kareem. Failure Recovery of World-altering Composite Semantic Services - a Two Phase Approach. In *Proceedings of the 14th International Conference on Information Integration and Web-based Applications #38; Services*, IIWAS '12, pages 299–302, New York, NY, USA, 2012. ACM.

[79] Jordy Sangers, Flavius Frasincar, Frederik Hogenboom, and Vadim Chepegin. Semantic Web Service Discovery Using Natural Language Processing Techniques. *Expert Syst. Appl.*, 40(11):4660–4671, September 2013.

[80] M. Schafer, P. Dolog, and W. Nejdl. An environment for flexible advanced compensations of Web service transactions. *ACM Transactions on the Web*, 2, 2008.

[81] N. Shadbolt, W. Hall, and T. Berners-Lee. The Semantic Web Revisited. *Intelligent Systems, IEEE*, 21(3):96–101, Jan 2006.

[82] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218 – 238, 2014.

[83] Doron Sherman. BPEL: Make Your Services Flow. Composing Web Services into Business Flow. *Journal in Web Services*, 3(7):16–21, 2003.

[84] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Guided Recovery for Web Service Applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 247–256, New York, NY, USA, 2010. ACM.

[85] E. Sindrilaru, A. Costan, and V. Cristea. Fault Tolerance and Recovery in Grid Workflow Management Systems. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 475–480, Feb 2010.

[86] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, et al. On the Enhancement of BPEL Engines for Self-Healing Composite Web Services. In *Proc. of the 2008 Int. Symposium on Applications and the Internet*, SAINT '08, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.

[87] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. Vision and challenges for realising the Internet of Things. *CERP-IoT. Cluster of European Research Projects on the Internet of Things*, 2010.

[88] Pawel Swiatek, Krzysztof Brzostowski, Jaroslaw Drapala, Krzysztof Juszczyszyn, and Adam Grzech. Development of Intelligent eHealth Systems in the Future Internet Architecture. In *Innovative Technologies in Management and Science*, volume 10 of *Topics in Intelligent Engineering and Informatics*, pages 73–94. Springer International Publishing, 2015.

[89] Wenan Tan, Leer Li, and Yong Sun. A Novel Performance Prediction Framework for Web Service Workflow Applications. In Qiaohong Zu, Bo Hu, Ning Gu, and Sopheap Seng, editors, *Human Centered Computing*, volume

8944 of *Lecture Notes in Computer Science*, pages 55–68. Springer International Publishing, 2015.

[90] M Tilly and Stephan Reiff-Marganiec. Fast Data Processing for Large-Scale SOA and Event-Based Systems. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 2015.

[91] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, M Eisenhauer, et al. Internet of things strategic research roadmap. *Internet of Things: Global Technological and Societal Trends*, 1:9–52, 2011.

[92] W3C. Web Services Architecture, 2004. Available at http://www.w3.org/TR/ws-arch/. [Online; Accessed: June 2015].

[93] Weidong Wang, Liqiang Wang, and Wei Lu. A Resilient Framework for Fault Handling in Web Service Oriented Systems. In *Web Services (ICWS), 2015 IEEE International Conference on*, pages 663–670, June 2015.

[94] Gerhard Weikum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[95] Quanwang Wu and Qingsheng Zhu. Transactional and QoS-aware dynamic service composition based on ant colony optimization. *Future Generation Computer Systems*, 29(5):1112 – 1119, 2013. Special section: Hybrid Cloud Computing.

[96] Ying Yin, Bin Zhang, Xizhe Zhang, and Yuhai Zhao. A Self-healing composite Web service model. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 307–312, Dec 2009.

[97] M. Younas, B. Eagelstone, and R. Holton. A Formal Treatment of a SACReD Protocol for Multidatabase Web Transactions. In *Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 899–908. Springer Berlin Heidelberg, 2000.

[98] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing Web services: issues, solutions, and directions. *The VLDB Journal*, 17:537–572, 2008.

[99] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 67–78, New York, NY, USA, 1994. ACM.

[100] Z. Zhao, J. Wei, L. Lin, and X. Ding. A Concurrency Control Mechanism for Composite Service Supporting User-Defined Relaxed Atomicity. In *The 32nd Annual IEEE Int. Computer Software and Applications Conf.*, pages 275–278, 2008.

[101] Zibin Zheng and Michael R. Lyu. An Adaptive QoS-aware Fault Tolerance Strategy for Web Services. *Empirical Software Engineering*, 15(4):323–345, August 2010.

[102] Zibin Zheng and M.R. Lyu. Selecting an Optimal Fault Tolerance Strategy for Reliable Service-Oriented Systems with Local and Global Constraints. *Computers, IEEE Transactions on*, 64(1):219–232, Jan 2015.

[103] Zibin Zheng, Yilei Zhang, and M.R. Lyu. Distributed QoS Evaluation for Real-World Web Services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 83–90, July 2010.

[104] Wei Zhou and Lina Wang. A Byzantine Fault Tolerant Protocol for Composite Web Services. In *International Conference on Computational Intelligence and Software Engineering (CiSE)*, pages 1–4, 2010.