Flexible Coordination based on the Chemical Metaphor for Service Infrastructures

Héctor Fernandez

N^o d'ordre : ANNÉE 2012

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informática

Ecole doctorale Matisse

présentée par

Fernández Héctor

préparée à l'unité de recherche (n^o + nom abrégé)
(Nom développé de l'unité)
(Composante universitaire)

Intitulé de la thèse: Coordination flexible fondée sur la métapthore chimique dans les infrastructures de services

Thèse soutenue à l'INRIA – Rennes le 20 Juin 2012

devant le jury composé de :

Jean-Pierre Banaâtre
Professeur d'Université, Université de Rennes 1 / président

Jean-Louis Giavitto
Directeur de Recherche, CNRS / IRCAM / rapporteur

Christian Perez
Directeur de Recherche, INRIA / rapporteur

Oscar Sanjuán Martínez
Professeur Associé,Université Carlos III de Madrid/ examinateur

Thierry Priol
Directeur de Recherche, INRIA / directeur de thèse

Cédric Tedeschi
Maître de Conférences, Université de Rennes 1 / co-directeur de thèse

Flexible Coordination through the Chemical Metaphor for Service Infrastructures

Abstract

With the development of the Internet of Services, composing loosely-coupled, and autonomous services dynamically has become one of the new challenges for large scale computing. While service composition systems are now a key feature of service oriented architectures, the coordination and execution of service compositions are still typically conducted through heavyweight centralized architectures, leading to problems related to scalability and reliability.

In a world where platforms are more and more dynamic and *elastic* as promised by *cloud computing*, service infrastructures are pushed toward more decentralized and dynamic interaction schemes. Addressing the characteristics of such platforms, nature-inspired, and in particular chemistry-inspired, analogies have recently regained attention in the search for flexible service coordination on top of dynamic large scale platforms.

In this thesis, we present a workflow management system able to solve a wide variety of workflow patterns in both a centralized and a decentralized way following the chemical model. Within this model, data and services are seen as molecules floating and interacting freely in a chemical solution. The decentralized workflow execution coordination is achieved through a set of reactions between those molecules. While its high expressiveness and adequacy for this context is being established, the chemical model severely suffers from a lack of proof of concepts. To tackle this problem, a prototype has been developed. Its implementation and evaluation in actual settings are discussed, establishing the viability of the concept, and thus opening the door to its future adoption.

Key-words: Service composition, service coordination, decentralization, workflow execution, workflow scheduling, nature-inspired models, chemical programming paradigm.

Résumé

Avec le développement de l'Internet des services, composer dynamiquement des services distribués faiblement couplés est devenu le nouveau *challenge* du calcul à large échelle. Alors que la composition de services est devenue un élément clef des plates-formes orientées service, les systèmes de composition de services suivent pour la plupart une approche centralisée connaissant l'ensemble des informations de flux de contrôle et de données du workflow, posant un certain nombre de problèmes, notamment de passage à l'échelle et de fiabilité.

Dans un monde où les plates-formes sont de plus en plus dynamiques, de nouveaux mécanismes de coordination dynamiques sont requis. Dans ce contexte, des métaphores naturelles, et en particulier la méthapore chimique, ont gagné une attention particulière récemment, car elles fournissent des abstractions pour une coordination flexible d'entités.

Dans cette thèse, nous présentons un système de gestion de workflow fondée sur la métaphore chimique, qui fournit un modèle d'exécution haut-niveau pour l'exécution centralisée et décentralisée de compositions (ou *workflows*). Selon ce modèle, les services sont vus comme des molécules qui flottent dans une solution chimique. La coordination de ces services est effectuée par un ensemble de réactions entre ces molécules exprimant

l'exécution décentralisée d'un workflow. Par ailleurs, si le paradigme chimique est aujourd'hui considéré comme un modèle de coordination prometteur, il manque des résultats expérimentaux. Ainsi, nous avons développé un prototype logiciel. Des expériences ont été menées avec des workflows d'applications réelles pour montrer la viabilité de notre modèle.

Mots-clés: Composition de services, coordination de services, décentralisation, exécution de workflows, ordonnancement de workflows, modèle inspiré par la nature, paradigme de programmation chimique.

Acknowledgments

Everything started in a sunny day on April 5th 2009, when I arrived at the capital of Brittany (Bretagne), a city called Rennes. I had gotten a phd position at the University of Rennes 1 to work on service orchestration using a chemical programming paradigm, a big challenge. Many doubts, feelings and thoughts passed through my mind during the first weeks. Nevertheless, I was enthusiastic, it was an opportunity to see how the things are going out of Spain, to meet new people (professionals and friends), to learn new methods, styles and a new culture from our good French neighbors. So, I said to myself "Challenge accepted".

Since 2009, many things, situations and people have crossed on my way to the finishing line. However, it is difficult to thanks to everybody in only 1 page, even more considering I am Spanish. I am gonna try to do my best and to go "Droit au but".

Firstly, I would like to thank the members of the team Paris → Myriads for your support, suggestions and coffee breaks. Even though we could not have a lot of feedback together due to the differences between our topics, I really learnt a lot with you guys. Thank you Christine, Alexandra, Amine, Julien, Roberto, Ghislain, Stefania, Loïc, Djawida, Maxence, Eugen, Jérôme, Pierre, Rémy, Guillaume, Piyush, Sylvain, André, Adrien, Pascal, Yvon, Bogdan, Marko, Cyril, Thomas, Amélie, Tylor, Katarzyna and Maryse.

Very special thanks to my advisor and co-advisor Thierry Priol and Cédric Tedeschi for your advice, help and professional orientation. Without your advices I could not be able to write this pocket book. I really learnt a lot from you everyday, even if Thierry was too busy the last year, you were there when I needed you (one problem → one article). Well, Cédric, "enfin" you were a great officemate but overall an awesome advisor, I am and I will be always your phd student. Working with you, it was an enriching experience that will help me a lot in my promising career.

Obviously, I do not forget all the members of my thesis defense committee. Thank you to Christian Perez and Jean-Louis Giavitto for reading and evaluating my thesis manuscript, I know you have a busy schedule. Gracias Jean-Pierre Bânatre por tu afecto, consejos y largas discusiones a cerca de temas de actualidad en mi país, y sobre todo por ser el presidente de mi comite. Also, I want to thank Oscar García Sanjuán to attend to my thesis defense coming from Spain (it is always a pleasure to work with you).

Generally, every success also comes with the support of your friends. I have to thank you guys for your anti-stress coffee sessions (at petit velo), soirees and sport activities, that we did together. I want to mention in this thesis some of my best friends which participated in this adventure: Andres, André, Lourdes, Laura, Cecile, Candice, Ionna, Kostas, Armando, Heverson, Or, Fer, Nolo, Santiago, Josip, Tyler, Marianne, Jorge, Bassy, Gasmus, Katarzyna, Cora, Armando and Ricardo. Evidently, I do not forget my bro – Marko Obrovac –, such a character, I could say we share many moments (goods/bads) together that makes us to become more than colleagues. I hope all these relationships I created with all of you, they will last forever.

Over the last two years, a person helped me in making things go, keeping my course in a right direction. It seems yesterday when we started to walk together but there is still a long path in front of us. Our long skype sessions will be over soon, you are very important for me and you will be always a ticket in my pocket, Dafni.

Por último, me gustaría agradecer a mí familia el apoyo que me han dado durante

estos tres años. No fue fácil al principio, sin embargo desde el primer día ellos siempre han estado ahí para darme fuerzas y sentirlos cerca, a pesar de la distancia. También, me gustaría mencionar a mis dos sobrinas Carolina y Alicia, dos soles de los que uno puede sacar fuerzas para superar cualquier obstáculo.

Contents

Introduction **13**

1 Service Oriented Computing **19**
1.1 Service Composition . . . 20
1.1.1 Requirements . . . 21
1.1.2 Service Interaction Protocols . . . 23
1.2 Orchestration . . . 24
1.2.1 Centralized vs Decentralized Orchestration . . . 25
1.2.2 Orchestration Languages . . . 26
1.2.3 Orchestration-based Workflow Management Systems . . . 31
1.3 Choreography . . . 35
1.3.1 Choreography Languages . . . 36
1.3.2 Choreography-based Workflow Management Systems . . . 38
1.4 Conclusion . . . 39

2 Flexible Models for Service Coordination **41**
2.1 Rule-based Models . . . 42
2.2 Tuplespace-based Models . . . 42
2.2.1 Preliminaries . . . 43
2.2.2 Existing Approaches . . . 44
2.3 Chemical Metaphor-based Models . . . 46
2.3.1 Gamma . . . 47
2.3.2 Similar Models . . . 48
2.3.3 Higher-Order Chemical Programming . . . 48
2.3.4 Chemistry-Inspired Models for Coordination . . . 54
2.4 Conclusion . . . 56

3 Decentralized Chemistry-Inspired Workflow Execution **59**
3.1 Architecture . . . 59
3.2 Molecular Composition . . . 62
3.2.1 Workflow Partitioning . . . 63
3.2.2 Chemical Workflow Representation . . . 64
3.2.3 Data Manipulation . . . 66
3.2.4 Chemical Rules for Distributed Execution . . . 68
3.2.5 Solving Workflow Patterns . . . 70
3.3 Execution Example . . . 78

3.4 Comparison to Existing Decentralized Approaches . . . 80
3.5 Comparison to Existing Composition Languages . . . 85
3.6 Conclusion . . . 86

4 Implementation and Experimentation 87
4.1 Software Prototype . . . 88
4.1.1 Architectures' Design . . . 88
4.1.2 Architectures' Implementation . . . 91
4.2 Performance Evaluation of HOCL-TS . . . 94
4.2.1 Workflows Considered . . . 95
4.2.2 Managing Large Workflows . . . 96
4.2.3 Exchanging Data . . . 97
4.2.4 Workflow's Complexity . . . 98
4.2.5 Discussion . . . 99
4.3 HOCL-TS vs HOCL-C . . . 99
4.3.1 Results . . . 99
4.4 Performance Comparison with Standard WMS . . . 101
4.4.1 Workflows Considered . . . 101
4.4.2 Centralized Experiments . . . 103
4.4.3 Decentralized Experiments . . . 104
4.4.4 Discussion . . . 105
4.5 Conclusion . . . 106

5 Decentralized Workflow Scheduling through a Chemically Coordinated System 107
5.1 Workflow Scheduling . . . 108
5.1.1 Scheduling Algorithms for Workflows . . . 108
5.1.2 Workflow Scheduler Systems . . . 111
5.2 Decentralized Shared Space for Workflow Scheduling . . . 112
5.2.1 Preliminaries . . . 112
5.2.2 Proposed Architecture . . . 114
5.2.3 Scheduling Molecules . . . 115
5.2.4 Chemical Nodes . . . 116
5.2.5 Meta-Molecules and Resource Retrieval . . . 118
5.2.6 Workflow Scheduling Process . . . 119
5.3 Evaluation . . . 122
5.3.1 Simulation Set-up . . . 122
5.3.2 Results . . . 123
5.4 Related Works . . . 125
5.5 Conclusion . . . 126

Conclusion 129

Bibliography 133

List of Publications 145

A General Concepts **147**
A.1 Elements of a Workflow Specification 147
A.2 Chemical Engine – Generic Rules 149
A.2.1 BlastReport Workflow Definition 151
A.2.2 Montage Workflow Definition 153
A.2.3 Cardiac Analysis Workflow Definition 153
A.3 Workflow Scheduling using HOCL 153
A.3.1 Workflow Decomposition 153
A.3.2 Opportunistic Load Balancing 158
A.3.3 Max-min 159
A.3.4 Min-min 159
A.3.5 Levelized Minimun Time 160
A.3.6 Heterogeneous Earliest-Finish-Time 161

List of Figures

1.1 Service composition example – Fire Damage Report –. 21
1.2 Orchestration *vs* choreography. 24
1.3 Centralized and decentralized orchestration. 26
1.4 A YAWL composition. 28
1.5 A Scufl composition. 29
1.6 Abstract and concrete workflow. 30
1.7 Pegasus workflow management system. 34

2.1 Tuplespace operations. 44
2.2 Service coordination using an XML tuplespace. 46

3.1 The proposed architecture. 60
3.2 Chemical workflow. 61
3.3 Different points of view of the architecture. 62
3.4 Molecular composition from an abstract workflow. 63
3.5 Decentralized workflow execution. 64
3.6 Simple workflow example. 65
3.7 Terms. 70
3.8 Parallel split. 71
3.9 Synchronization. 71
3.10 Exclusive choice. 72
3.11 Discriminator. 73
3.12 Simple merge. 74
3.13 Synchronization merge. 75
3.14 Multi merge. 76
3.15 Cancel activity. 77
3.16 Example of coordination. 79
3.17 Workflow execution, steps 0-3. 81
3.18 Workflow execution, steps 4-7 82
3.19 Workflow execution, steps 8-12. 83
3.20 Message passing between engines. 84
3.21 Tuplespace data and messages passing control. 85

4.1 Differences between architectures. 88
4.2 HOCL-C WMS architecture. 89
4.3 HOCL-TS WMS architecture. 90
4.4 HOCL-P2P WMS architecture. 91

4.5 HOCL-C implementation. . . . 92
4.6 HOCL-TS implementation. . . . 93
4.7 HOCL-P2P implementation. . . . 94
4.8 30-task workflow. . . . 95
4.9 60-task workflow. . . . 96
4.10 100-task workflow. . . . 96
4.11 Performance results, complexity of workflows and services. . . . 97
4.12 Performance results, data exchange. . . . 98
4.13 Discriminator and Parallel split pattern tests. . . . 100
4.14 Performance on basic patterns. . . . 100
4.15 Performance on advanced patterns. . . . 101
4.16 BlastReport and Cardiac workflows structures. . . . 102
4.17 Montage workflow structure. . . . 103
4.18 Performance results, Montage. . . . 103
4.19 Performance results, BlastReport. . . . 104
4.20 Performance results, CardiacAnalysis. . . . 104

5.1 Overview of the proposed architecture. . . . 114
5.2 Two-layer architecture. . . . 115
5.3 Data molecules. . . . 117
5.4 Chemical node. . . . 117
5.5 Two DHT layer. . . . 118
5.6 Workflow decomposition. . . . 120
5.7 Multiple workflow mapping. . . . 122
5.8 Execution time. . . . 124
5.9 Network traffic. . . . 124
5.10 Traffic per node. . . . 125

Introduction

Since the emergence of computer science, the quality of applications' design has been a subject of considerable research. Hence, the design of applications has been constantly evolving over time, addressing different requirements. Increasing modularity and flexibility, it evolved from procedural-based to object-based (bundling together procedures and data structures) standalone applications. With the appearance of Internet, applications became distributed. Applications started to rely on a composition of different processes located on distinct computers connected in a network. Since then, the design of distributed applications has also evolved, in terms of reusability, dynamicity or modularity, moving from RPC-based (Remote Procedure Call), to object-based (Remote Object Invocation) to component-based applications. Applications are composed of different components communicating together using technologies such as DCOM (Distributed Component Object Model) [41] and CORBA (Common Object Request Broker Architecture) [119]. However, these technologies suffered from tight coupling, poor dynamicity and interoperability, issues at the origin of the development of a new paradigm called *service oriented computing*. In this paradigm, applications are conceived as a compound of modular, loosely coupled and reusable units of functionality called *services*. The resulting architectures built by the combination of services are known as *service oriented architectures* (SOA). SOA can be seen as a blending of the older concepts of *distributing computing* and *modular programming*.

The success behind SOA is the interoperability, reusability and flexibility (dynamicity and loose coupling) of the services combined.

Consequently, with the blooming of Web Services and the appearance of powerful integrated development environments (IDE) such as Eclipse [1] and MS Visual Studio [2], and tools such as Apache Tomcat [3] and Apache Axis [4], the development of SOA applications has been made much easier than with RMI (Remote Method Invocation) [58] and CORBA technologies. Web services are independent units of functionality that expose their capabilities to be easily accessed, used, and reused by anyone. As a result, applications tend more and more to take the shape of compositions of independent, network-enabled services bounded at run time.

These compositions allow to build more complex applications represented by a temporal composition of distributed services, commonly called *workflow*. The success of the *myExperiment* [5] platform for service and workflow sharing is a clear sign of the success

[1] http://aws.amazon.com/eclipse/
[2] http://www.microsoft.com/visualstudio/en-us
[3] http://tomcat.apache.org/
[4] http://axis.apache.org/axis/
[5] www.myexperiment.org

of this shift. Using the *myExperiment* platform, users can discover, share and reuse workflows from other users, at the same time promoting new collaborations.

Similarly, the recently emerged platform paradigm known as *cloud computing* also conceives aspects related to the infrastructure (IaaS), platform (PaaS) and other higher level functionalities (SaaS) as services, again showing the importance of this shift of paradigm. IaaS exposes as services a variety of equipments used to support operations such as storage, hardware, servers and networking components, like the well-known Amazon EC2 [6]. PaaS provides an environment for development, deployment and management of applications during its whole life cycle, as does for instance GoogleAppEngine [7]. Finally, SaaS exposes applications with diverse functionalities as services. This trend can be experienced in every day life for instance with utilities like Gmail [8], Mobile Games and gadgets.

Motivation

The growing complexity of service-based applications gives birth to large-scale service compositions, in which participants from different organizations collaborate in order to achieve a common goal. The term *complexity* encompasses a combination of different factors: the number of services involved, the computation/storage demand, the volatility of services involved and the coordination structures for the most important. According to the characteristics of the service interactions, current service coordination models do not seem to be the most appropriate solution for handling these compositions.

The volatile nature of service interactions requires dynamic and loosely coupled service coordination models that provide the ability to quickly adapt to changes, *i.e.*, providers do not continuously supply their services. The existing workflow languages exhibit some limitations with respect to *flexibility*: even though some of them support dynamic service binding, they lack adaptation mechanism regarding changes in the composition structure.

Another issue of current service infrastructures is that they are built upon highly centralized workflow engines exposing several weaknesses when processing large-scale compositions. First, they generally suffer from poor scalability and low reliability, central workflow engines being potential processing and communication bottlenecks as well as single points of failure [13]. In a more societal point of view, they also raise privacy/security issues, as all data and control passing through central engines and repositories could for instance leave the door wide open for industrial espionage.

As a consequence, it becomes crucial to promote a decentralized vision of service infrastructures, as for instance suggested in [142]. The benefits of a decentralized approach are manifold. First, as the coordination and data are distributed among a set of nodes, there is no single point of failure. No central engine acts as a potential bottleneck, network traffic is reduced, and the approach is globally more scalable. Second, the direct and asynchronous fashion of communications between two nodes involved in the composition (without the need for central coordination) brings better throughput and graceful degradation in case of failure [45]. Finally, no central engine takes control over

[6] http://aws.amazon.com/ec2/
[7] https://developers.google.com/appengine/
[8] https://mail.google.com/

data and work, each node integrating a local workflow engine, and having only a partial view of the composition.

Nevertheless, another limitation comes up when defining workflow specifications to be executed in a decentralized manner. The existing workflow management systems use *centric-based* and *low-level abstraction* languages (often proprietary languages) which do not provide adequate abstractions to express a distributed execution naturally. Among the different solutions to decentralize the workflow execution, none of them is better suited than others. While some works [28, 64] proposed different techniques to partition the workflow definition prior to the execution, other approaches [19, 148] prefer to forward the whole workflow definition along all the participants. Workflow partitioning is a complex task, to be done statically at design time. To sum up, there is a need for simpler and higher level abstraction models to be able to define workflows intended to be executed in a decentralized way.

Over the decades, computing platforms evolved a lot in computing power and storage capacity, facing the ever growing demand of applications. Recently, cloud computing and its elasticity (the capacity to add and remove new computing resources on demand) makes this platform an appealing tool for the processing of these large-scale service interactions. For instance, the Magellan project [9] aims at providing a cloud-based platform for scientists. However, the computing power demanded by some service-based applications, especially scientific related to astronomy [9] and bioinformatics [10], can reach such proportions that some resource providers alone are not able to face them. This led to collaborations amongst different resource providers, giving birth to large *resource federations*, in particular for *community clouds* [87]. It allows different cloud providers to share their resources, making it possible for these workflows to be deployed and executed. The idea of sharing resources, however, carries with it new challenges such as the interoperability between different resource providers, elasticity, security and economical issues [44, 56]. So, our model should be able to deal with such platforms giving the ability to 1) take resources into account in the workflow by injecting some scheduling mechanisms, and 2) run in a fully decentralized fashion.

Contribution

To tackle the problems previously presented, this thesis explores an unconventional programming model for the management of service coordination.

A Chemistry-Inspired Model for Service Composition

Recently, nature metaphors, and in particular chemistry-inspired analogies, have been identified as a promising source of inspiration for developing new approaches for autonomous service coordination [135]. Among them, the *chemical programming paradigm* [21] brings some interesting features like the autonomous behavior and the high level of abstraction. Within such a model, a computation is basically seen as a set of reactions consuming some molecules of data interacting freely within a *chemical*

[9] http://irsa.ipac.caltech.edu/

[10] http://epigenome.usc.edu/services/nextgen/data_recovery_analysis.html

solution producing new ones (resulting data). Reactions take place in an implicitly parallel, autonomous, and decentralized manner. More recently, the Higher-Order Chemical Language (HOCL) [22] raised the chemical model to the higher-order, providing a highly-expressive model: every entity in the system is seen as a molecule. Moreover, rules can apply on other reaction rules, programs dynamically modifying programs, opening doors to dynamic adaptation.

Lately, it has been shown that such a paradigm is well-suited to express service orchestration [27], and describe the enactment of workflows engine [99]. Thus, to model service interactions, we leverage the chemical analogy and extend it to *molecular composition* to model the decentralized execution of a wide variety of workflow structures, a.k.a, workflow patterns [132]. Following this *molecular composition analogy*, services, as well as their control and data dependencies, are molecules interacting in the workflow's chemical solution. Chemical rules (higher-order molecules) are in charge of its decentralized execution, by triggering the required reactions locally and independently from each others. These local reactions, together, realize the execution of the specified workflow.

Towards a Chemistry-Inspired Middleware

The first part of this thesis intends to show that this model is a simple and appropriate solution to execute workflows in a decentralized way. It also establishes its expressiveness and adequacy to service coordination. The second part of the thesis intends to address another essential problem: the actual experimentation of the chemical model has remained quite limited until now. There is a strong need of a proof of concept to show its viability, in particular compared to current reference workflow management systems. Based on the abstractions developed before, we present a workflow management system able to solve a wide variety of workflow patterns both in a centralized and a decentralized way. Its implementation and performance evaluation on different real workflows are discussed, establishing the viability of the concept, and lifting a barrier on the path to its future wider adoption.

In order to complete this chemistry-inspired system and take resources of the underlying platform into account, we extend our model, and finally propose a fully decentralized workflow scheduling framework, which solve the current drawbacks when scheduling workflows in a *resource federation*, in particular for *community clouds*. This framework for just-in-time scheduling is organized in two layers. The top layer is a chemically-coordinated shared space where workflows are decomposed into tasks, which are mapped to resources following simple chemical rules. The bottom layer implements this shared space in a fully decentralized way, based on a peer-to-peer overlay network allowing the efficient storage and retrieval of molecules.

Organization of the Thesis

In Chapter 1, the principles and methodologies for the development of service-based applications are introduced. The most used methodologies for the modelling of service interactions, as well as their executable languages and implementations are described, detailing the benefits and drawbacks inherited by using each methodology.

Chapter 2 introduces more flexible models to coordinate service compositions, in

particular dealing with loose coupling and dynamicity. The purpose of this chapter is also to give all the needed information to understand the demand for new service coordination models, as well as to introduce its main features and the language built atop of the chemical metaphor. Its adequacy to model service composition is also discussed.

Chapter 3 and Chapter 4 present the main contribution of this dissertation [11]. We describe, in Chapter 3, a decentralized system for the execution of workflows based on the chemistry-inspired coordination model. This chapter introduces a new analogy for service composition, namely *molecular composition*. This analogy will allow to model a wide variety of workflow structures by composing molecules representing the service interactions. Consequently, in Chapter 4, we focus on the architectural design, the implementation and the experimental validation of this chemistry-inspired workflow management system. In particular, the experimental campaign includes the validation of this system in comparison with the more representative and used workflow management systems.

Finally, in Chapter 5 [12], we focus on workflow scheduling, in particular, in the decision making process for mapping each job of the workflow to the most appropriate resource in a *community cloud* infrastructure. The scheduling framework proposed extends the concept presented in Chapter 3 and provides a fully decentralized *just-in-time* workflow scheduling system. A simulation-based evaluation of the performance and network overhead of the framework is also discussed.

[11] I am the main author of the work presented in Chapter 3 and Chapter 4
[12] I am one of the authors of the work presented in Chapter 5

Chapter 1

Service Oriented Computing

Initially, the Web was intended for human use, however most experts agree that it has evolved, with the proliferation of modular services, to an autonomous system. Nowadays, this autonomous system can be called by programs instead of humans [35, 92]. A service can be defined as a self-contained software utility that exposes a capability or information as a reusable unit. Services provide a high level of abstraction which is required for organizing applications at large-scale. Hence, the development of applications based on the combination of services produces an improvement in the productivity and quality of these systems. This trend gave birth to a new paradigm known as service oriented computing (SOC).

In service oriented computing, developers use services as building blocks for the development of applications. In order to operate in a SOC environment, services must expose their functionalities and properties in a standard, machine-readable format. SOC is required to offer three capabilities: description, discovery, and communication. Web services represent a good SOC example: developers implement these capabilities using Web Services Description Language (WSDL) [2] (for description), Universal Description, Discovery, and Integration (UDDI) [5] (for discovery), and SOAP [1] or REST [65] technologies (for communication). The correct combination of these three capabilities allow to satisfy the primary goal of SOC, *i.e.*, the creation of collections of services accessible on Internet via standardized protocols, and whose functionality should be automatically discovered and integrated into applications to create more complex services. In such a way, companies can design complicated transactions that involve multiple services from different enterprises in a complex invocation chain [48]. Service oriented computing based on Web services is currently considered as one of the main drivers for the software industry [35, 92]. The World Wide Web Consortium (W3C) [1] defined Web services as follows:

> *'Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols'.*

[1] http://www.w3.org/TR/ws-arch/

Therefore, SOC re-invents the way enterprises work together: common tasks in a business process or supply chain can be easily outsourced to external service providers by improving performance, enabling cost savings and increased flexibility.

In this chapter, we distinguish the main formalisms for service composition which will help us to better understand the SOC paradigm. Section 1.1 introduces the service composition concept and the existing approaches for modelling the service coordination. Section 1.2 describes the most used and mature methodology for service composition called *orchestration*. Section 1.3 presents the other well-known, although less, used approach for service composition called *choreography*. Finally, Section 1.4 draws some conclusions.

1.1 Service Composition

There are many ways to describe service oriented architectures, however the 'LEGO' analogy is considered as one of the most appropriated. In this analogy, each 'LEGO' block is seen as a concrete service, whose size and shape properties allow to distinguish it from other services. Therefore, based on the 'LEGO' analogy, **service composition** can be seen as a combination of different blocks representing every day life objects.

To create applications, developers use service composition [94, 122], which they introduce on top of SOC's capabilities. Developers and users can solve more complex problems by combining available services and ordering them to best suit their problem requirements. A rapid application development is achieved using service composition, as services can be dynamically discovered and reused to design new applications. This allowed the service oriented computing paradigm to become one of the dominant approach to design distributed applications. However, these applications can often present some problems, such as a poor scalability and a high dependability. In fact, there is no well-defined specifications to determine which requirements a service composition must satisfy. Thereby, a good service composition is sometimes an art form, building architectures by using certain strategies. The success of these strategies depends on specific use cases and a number of different factors.

The common mechanism of collaboration between a service provider and a consumer can be defined in two steps:

1. The service provider publishes a service S_A.
2. The service consumer discovers and invokes the service S_A and waits until its execution is completed. Finally it uses the result of this execution.

In Figure 1.1, a group of services are composed to model the actions taken by the maintenance department of a mall center after a small fire. If the 'Evaluate Damage' service determines that the fire caused structural damage, the 'Notify Fire Dept.' service will contact the fire department. If the 'Evaluate Damage' service also determines that the damage exceeds 2000 euros, the 'Notify Insurance' service will contact the insurance company. In both cases, the 'Submit Report' service will create a report which will be submitted to the building manager. This example shows through the service composition, four services with independent and well-defined functionalities can be easily combined to build a more complex application, *i.e.*, *Fire Damage Report*.

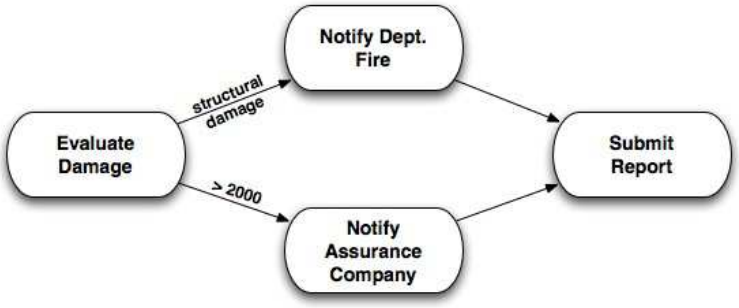

Figure 1.1: Service composition example – Fire Damage Report –.

Following the SOC paradigm, services can be seen as 'black box' since the implementation details are hidden to the consumers. Consumers only have access to their interface descriptions. Since the services are accessible on Internet, the service composition is named **Web Service Composition**, and the resulting aggregation of services is referred to as a *composite Web service* or *workflow*. A workflow is a directed acyclic graph (DAG) that specifies the (directed) dependencies between services composed.

To sum up, service composition offers reuse possibilities and access to a variety of complex services. In order to accomplish these benefits, we first need to define the requirements when composing services.

1.1.1 Requirements

Service composition builds on the use of flexible services and interaction mechanisms to compose them. To achieve that, services are decomposed into a service hierarchy based on functionality, by abstracting the layers, and by defining compositions that can be separated from the makeup of the services themselves, as mentioned in [113].

A composition system must therefore satisfy several requirements:

- **Connectivity:** every part in a composition must guarantee the connectivity. The service connectivity can be defined as the availability of a service to interact in a composition. With reliable connectivity, we can analyze the data and control dependencies among services, what facilitates the validation of a service composition.

- **Correctness:** composition correctness requires verification of the properties such as QoS properties, *i.e.*, dependability, timelines and security.

- **Scalability:** composition systems must scale with the number of involved services in a workflow. Centralized composition systems manage data and control dependencies among the involved services, what may produce communication and performance bottlenecks.

However, a good composition should not only satisfy these requirements but also services of the composition themselves should have the following characteristics:

- **Modularity and granularity:** Based on service oriented architectures, applications are decomposed into modular services which are self-contained. Granularity defines the functional richness for a service, the more coarse-grained a service is, the richer or larger the function offered by the service is. Coarse-grained services provide a greater level of functionality within a single service operation, by reducing complexity, network overhead and reusability. In contrast, fine-grained services exchange small amounts of information to complete a specific discrete task, by increasing the reusability. Note that, a task is a workflow operation which can represent the invocation of a Web service, an external command, or a database operation. Even though the level of granularity generally depends on the purpose of the software entity, SOA tends to build coarse-grained services.

- **Encapsulation:** There is a strict separation from the public interface of a service and its internal implementation details, which is seen as a 'black box'.

- **Loose coupling:** Coupling describes the number of control and data dependencies between two services in an interaction. Aspects such as *flexibility* and *extensibility* of a system can be affected depending of the degree of coupling between services. In general, we can distinguish between loosely and tightly coupled services. Tightly coupled services have many known and, what is more importantly, hidden dependencies. While loosely coupled services have few, well-known and well-managed dependencies.

- **Isolation of responsibilities:** Services are responsible for the execution of discrete tasks or the management of specific resources. This provides one place for each function to be performed, providing consistency and reducing redundancy.

- **Reuseability:** Services participating in a composition can be deployed and modified independently from each other. Requirements such as modularity, encapsulation, loose coupling and isolation of responsibilities enable services to be used into multiple compositions or accessed by multiple service consumers.

- **Dynamic discovery and binding:** Services can be discovered and invoked at run-time through the use of a service repository [113]. The dynamic binding of services increases the loose coupling degree of a system.

- **Stateless:** Service operations are stateless. This means that each time a service is invoked, a new instance will be created. Stateless services provide better flexibility, scalability and reliability. However, they need to include additional information in every request in comparison with stateful services. In contrast, stateful services should be able to save their state in a persistent data source and/or maintain the state information between invocations.

- **Self-describing:** The service descriptor files provide a complete description of the service interface, its operations, the input and output parameters. Similarly, pre-conditions, post-conditions and constraints about the operations may be also described.

- **Composable:** Services themselves can be composed from other services, and can be mixed and matched as needed to build more complex services.

- **Governance:** Relationships between service consumers and providers establish the contracts in a composition. A contract describes the policies to be accomplished by participants in an interaction. This technique is sometimes referred to as Service Level Agreements (SLAs) [128].

- **Location/language/protocol-independent:** Services are designed to be independent of communication protocols, platforms and physical locations in which they are deployed.

1.1.2 Service Interaction Protocols

In service composition, the collaboration between services or service interactions can be defined in two different manners: orchestration and choreography. Orchestration and choreography approaches are two methodologies, which define the control and data dependencies, contracts and policies among involved services in a workflow. However, orchestration and choreography focus on different aspects in a composition.

Orchestration describes a control and data flow among services from the perspective of one participant acting as a coordinator node called *the orchestrator* [106, 113]. The point of reference for orchestration is a single orchestrator. As shown by Figure 1.2 (left side), the orchestration describes all the data and control dependencies between the tasks (rounded rectangles) and participants (rectangles), as well as other execution details, such as condition checking (rhombus). While choreography gives a full perspective – global contract – of the behavior of all the involved participants [29, 91, 113]. In contrast, as shown by Figure 1.2 (right side), the choreography describes the sequence of messages involving the two participants, where each participant collaborates to coordinate the whole composition (the control is decentralized).

To sum up, both methodologies describe a workflow. Orchestration defines a model which can be directly executed, whereas choreography is purely an agreement among services in a workflow, modelling how a given collaboration should occur.

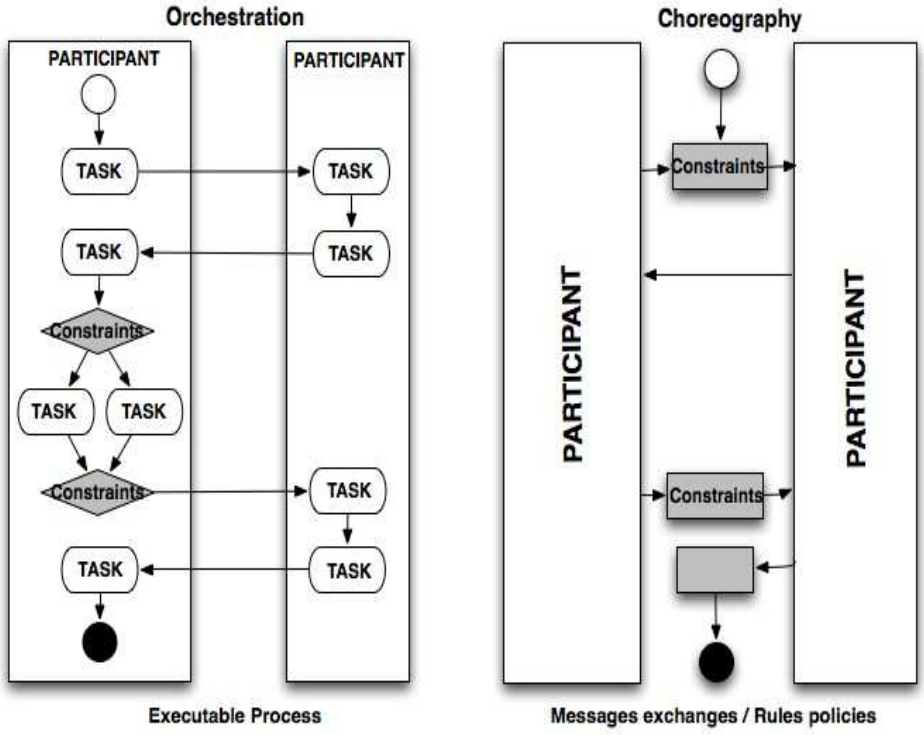

Figure 1.2: Orchestration *vs* choreography.

Over the last few years, there were passionate debates between developers to decide which methodology to use. Orchestration and choreography are widely used for composing service-based applications [91]. However, the orchestration community has the most traction in the standards bodies and in technology adoption. In fact, the software industry mainly uses a standard orchestration language to model the business process of enterprises called BPEL [101]. The BPEL language is introduced in Section 1.2.2.1. In contrast, the number of approaches executing choreography definitions are still growing day after day without following any standard. In our work, we present a service coordination system that can be defined as a trade off between the two methodologies. A choreography model is defined among orchestration engines, to achieve a decentralized workflow execution.

The rest of the chapter is organized as follows. Section 1.2 and Section 1.3 details the orchestration and choreography interaction protocols respectively. Finally, Section 1.4 summarizes the concepts and ideas presented in this chapter.

1.2 Orchestration

In the following, we focus on orchestration architectures and languages due to the widespread adoption and ripeness of this model.

1.2.1 Centralized vs Decentralized Orchestration

The execution of a service orchestration can be coordinated using two types of architectures: centralized and decentralized.

The most common approach to orchestrate services is to use a centralized engine. This approach is simple for management and monitoring. Workflow management defines and verifies that a set of tasks produce a certain result, and monitoring displays the status information about completed and currently executing workflows. In centralized approaches, a workflow is executed by a single coordinator node, *the orchestrator*. It receives the client requests, makes the required data transformations and invokes the services based on the control-logic previously defined. Hence, in Figure 1.3 (left side), the execution of the workflow relies on the orchestrator responsible for coordinating data and control flows between services S_1, S_2, S_3 and S_4. During the actual execution of the workflow, the orchestrator first invokes S_1 by sending a message to node '$res-2.domain.fr$', then waits for the result of S_1 (sent by '$res-2.domain.fr$'), and finally invokes S_2 and S_3. In a centralized orchestration, all data are transferred among services using the orchestrator instead of being transferred directly from one service to another. However, centralization leads to some performance limitations and bottlenecks [113, 144], what limits its reliability for the execution of computation and data intensive workflows.

On the other hand, in decentralized approaches, each participating service is responsible for partial orchestration (managing its control and data dependencies), based on its individual rules without any central coordination. The participants collaborate to execute a workflow with every one executing a part of it, leading to the following benefits [45]:

1. There is no centralized coordinator, eliminating potential bottlenecks.
2. The data is distributed among engines, what reduces network traffic and improves transfer time.
3. The control-logic is executed in parts on several engines, which improves concurrency.
4. A communication mechanism based on asynchronous message exchanges between engines, brings a better throughput and graceful degradation in case of failure.

An example of decentralized orchestration is illustrated on Figure 1.3 (right side). The execution of the workflow relies on the collaboration between four workflow engines, *i.e.*, one engine per involved service. Workflow engines must be aware of the workflow specification, when to invoke their operations, and how to interact with the other engines, through messages. Hence, in the workflow, nodes '$res-2.domain.fr$', '$res-1.domain.fr$' and '$res-34.domain.fr$' may communicate directly (rather than through a coordinator node) to transfer data and control when necessary (*i.e.*, after S_1 finishes).

The decentralized orchestration appears as the only way to compose large-scale workflows, where data move only in a given direction based on the data-flow constraints through different environments. Similarly, the time wasted with the network latency to communicate with the different engines is regained by reducing the workload of a central engine. The term *workload* is assumed to be the amount of work that the workflow engine has been given to do at a given time.

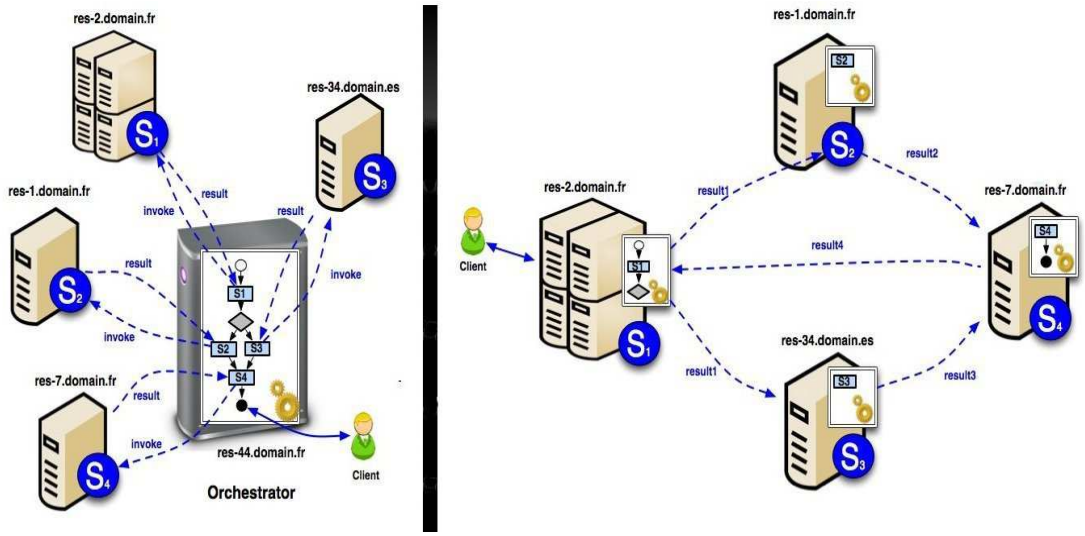

Figure 1.3: Centralized and decentralized orchestration.

Nowadays, there is a wide literature about decentralized approaches to execute workflows [45, 88, 93, 121, 145].

In [45, 93], a hybrid model commonly uses multiple centralized engines, where tasks are, in advanced, planned to be distributed across orchestration servers. The data and control dependencies among services are statically analyzed, and then the workflow definition is partitioned and distributed to each server. In contrast, these approaches lack of adaptation to the changes in the environment, workflow are partitioned in parts at build-time and then distributed to each server. Recently, more dynamic solutions have been proposed [145]. In [145], the authors proposed a continuation-passing mechanism where information on the remainder of the execution is carried in messages to each engine involved in the coordination.

More recently, other approaches use a shared space as storage mechanism for control and data information, and as a communication mechanism among services [88, 121]. Using a shared space as a communication infrastructure, the control and data dependencies can be exchanged among participants through this space instead of directly between them. These approaches, which improve loose coupling, will be detailed in Chapter 2.

1.2.2 Orchestration Languages

In this section, we introduce some of the most mature and used orchestration languages for service composition in both business and scientific domains.

1.2.2.1 WS-BPEL

WS-BPEL (Business Process Execution Language for Web Services) [101] is the *defacto* standard for service orchestration in business domains. WS-BPEL (in short BPEL) is an XML-based executable language that has emerged from the earlier proposed XLANG [113] and Web Service Flow Language (WSFL) [3]. It enables the construc-

	YAWL	SCUFL	BPEL	GWorkflowDL	DAX
Type	Control-data driven	Data driven	Control-data driven	Control-data driven	Data driven
Parallel execution	Explicit	Implicit	Explicit	Explicit	Implicit
Abstract workflow	–	X	–	X	X
Platform details	–	–	–	X	X
Dynamic binding	–	–	X	–	–

Table 1.1: Comparison among languages.

tion of complex business processes from existing Web services, which can also represent other business processes. Thus, this language enhances the reusability by exposing BPEL processes as Web services.

BPEL is an imperative and control-based workflow language by including the explicit definition of the processing order. Web services are primitive execution blocks, and service composition is achieved using control primitives (sequences, parallels, conditionals, and loops). The primitive execution blocks cover *invoke, receive* and *reply* operations which enable asynchronous or synchronous service invocation. On the other hand, the control primitives are used to build complex structures. More precisely, the *sequence* primitive offers the ability to define ordered sequences of tasks, *flow* executes a collection of tasks in parallel whereas the execution order is given by links between the tasks. The *switch* control primitive allows branching, *pick* allows to execute one of several alternative branches and *loops* can be defined using the *while* primitive. In addition, there is another set of primitives for managing events such as the *wait* primitive, which awaits during some time, the *terminate* primitive stops the execution of the workflow instance, the *assign* primitive copies data from one message to another, the *throw* primitive announces errors, and the *empty* primitive does nothing emulating an empty operation.

BPEL includes the feature of *tasks scoping* that enables the modelling of sub-processes inside of an existing one forming a tree-like structure composed of BPEL processes. In addition, it is possible to specify fault handlers and compensation handlers for scopes. Faults handlers get executed when exceptions occur, for instance, through the execution of the mentioned *throw* primitive. Compensation handlers are activated when faults occur or when *compensation* primitive that force compensation of a scope are executed.

Even though originally designed for business workflows, BPEL has recently gained a lot of attention by the scientific community, mainly because of the appeal of the Service Oriented Architecture (SOA) paradigm in this domain. However, the modelling of scientific data-flows using BPEL is still a tedious experience [125].

1.2.2.2 YAWL

Some contemporary workflow management systems are based on Petri Nets [74], a widely used model of discrete systems invented in 1939 by *Carl Adam Petri*. However, they lack support for expressing some workflow patterns (*i.e.*, involving multi-instances or complex synchronizations), or require a lot of specification effort, as pointed it out by YAWL's authors [131]. A workflow pattern is a formal way of describing a solution to a design problem that appears repeatedly when defining a workflow structure. These patterns

need to be mapped onto High Level Petri Nets (HLPN) [79], where the term HLPN is referred to Petri Nets extended with color, time and hierarchy.

YAWL [131] is a workflow language based on Petri Nets but extended with additional features to enable a more intuitive workflow patterns identification, what reduces the complexity of modelling workflows. These features use HLPN, and thus support workflows with multiple instances, composite tasks, OR-joins, removal of tokens, and direct transitions.

A workflow specification using YAWL consists of a composition of tasks (either atomic or composite) that form nets with a tree-like structure. Each atomic task represents the leaves of this hierarchy distribution. Composite tasks or subprocesses refer to the different sub-nets or sub-levels in this tree, as suggested in Figure 1.4. There is one net that is not referred by other composite tasks: it forms the root of this tree workflow. The nets are also composed of another element called conditions, which can be interpreted as places. Unlike Petri nets, it is possible to connect 'transition-like objects' such as composite and atomic tasks directly to each other without using a 'place-like object' like conditions between them.

An example of a YAWL workflow representation is illustrated on Figure 1.4. It shows the different elements of a YAWL tree-like structure: the *root* representing the whole workflow, atomic tasks (squares), a composite task as a subnet scoping another subprocess, and the conditions (circles).

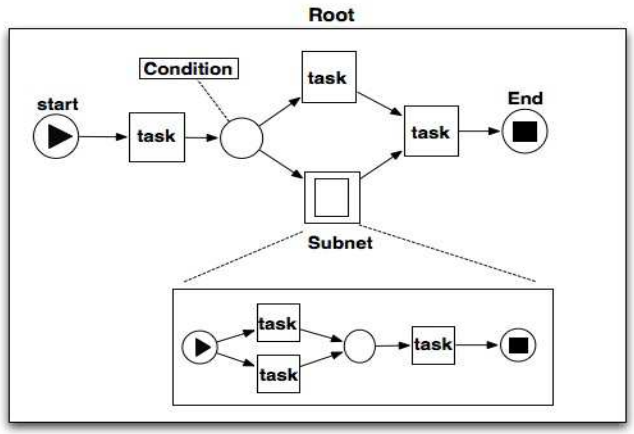

Figure 1.4: A YAWL composition.

1.2.2.3 SCUFL

Scufl (Simplified Conceptual United Flow Language) [125, 129] is an XML-based and data-driven workflow language developed by the myGrid consortium[2], which aims at contributing to the development of services-based toolkits for eScience. Scufl is used by Taverna Workflow Management System [104] to express scientific workflows, which is detailed later, in Section 1.2.3.2. It enables the definition of data-flows between different local and remote services provided by external applications. Similarly, the definition of

[2]http://www.mygrid.org.uk/

some control flow patterns can be modeled through control links between tasks. However, this coordination constraint can be only modeled when existing a data dependency between involved tasks.

In Scufl, the tasks of a workflow are called *processors*. A processor receives data on its input, execute some operations and produces data on its output. Processors represent Web services or other executable components. Using data links, processors are composed with each others, defining data dependencies between the output of a processor and the input of another. In addition to these data links, Scufl allows the definition of control links among processors that specify precedence conditions, *i.e.*, a processor can execute only when another one has successfully completed its execution. The inputs of a workflow are represented as sources, and the final results are represented as sinks. The execution of a workflow starts from the sources and finishes when all sinks have either produced their output or failed.

An example of a Scufl representation is illustrated on Figure 1.5. It is composed of three sources associated to two processors ($Task_1$ and $Task_2$) through data links, and one sink representing the end of the execution. According to the data links, once the sources are available, the processors will be invoked, transferring its final result to the sink.

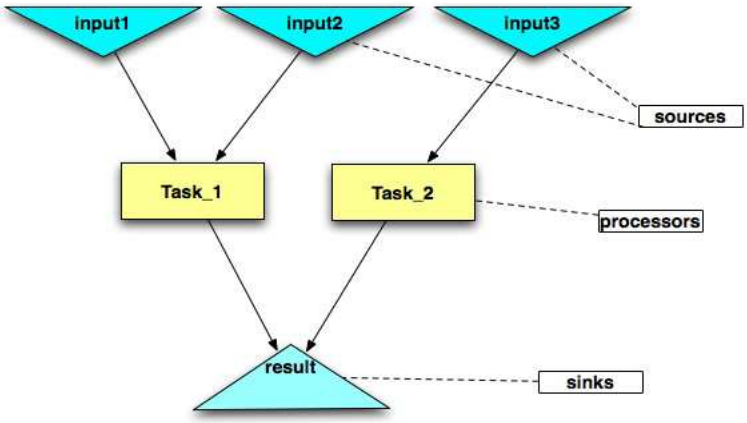

Figure 1.5: A Scufl composition.

In Taverna, a processor definition includes information related with a fault tolerance mechanism, that calls an alternative service if the service of first choice, after a certain number of retries, fails. Thus, this information specifies: the implementing services, the number of retries, the time between retries, and optionally an alternative service.

Nowadays, Scufl is one of the more mature and used workflow languages to model and execute scientific data-driven applications.

1.2.2.4 DAX

DAX [55] is an XML-based DAG specification used by Pegasus Workflow Management System [55] to express *abstract* workflows (see Section 1.2.3.4 for a description of Pegasus). An abstract workflow offers a global view of a workflow specification without giving

execution details, *i.e.*, platform independent. Nevertheless, DAG could also provide a concrete workflow specification by explicitly specifying the data location and resource description in the workflow specification.

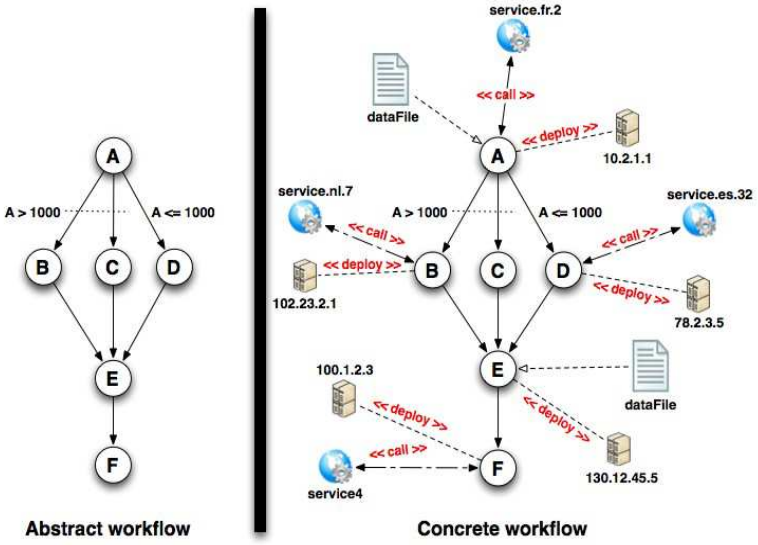

Figure 1.6: Abstract and concrete workflow.

The differences between *abstract* and *concrete* workflows are illustrated on Figure 1.6. In a concrete workflow, the specification contains enough details to be actually executed, such as the candidate services to invoke (A calls the service '$service.fr.2$'), the resources on which to deploy the tasks (B is deployed on the resource '102.23.2.1') and the data files to use (E processes the file '$dataFile$'). Nevertheless, in an abstract workflow, the specification contains enough details for its logic to be understood by anyone, as suggested by Figure 1.6 (left side).

A DAX specification consists of three parts: the first part describes the input/output data files that will be consumed and produced by tasks within the workflow; the second part specifies the arguments of each task or jobs such as the name, the name of input and output files that were previously defined; the third part defines the data dependencies between the tasks. The relationships are defined as child parent relationships with no cyclic dependencies.

1.2.2.5 GWorkflowDL

The GWorkflowDL (Grid Workflow Description Language) [15] is an XML-based language for representing Grid workflows based on High Level Petri Nets (HLPN). This language is currently the basis for the Java Grid platform [14] of the University of Muenster in Germany and the K-Wf Grid European Project [3], which addressed the need for a better infrastructure for the future Grid environment.

[3] http://cordis.europa.eu/projects/71832_en.html

GWorkflowDL is a control and data driven language, that uses the HLPN concept of edge expressions to assign a concrete service to a transition, and conditions as a mechanism to define the control flow relationships. In such a way, the original HLPN model is neither modified nor extended to describe service composition.

A workflow definition using GWorkflowDL presents two abstraction levels:

- The first level, called *generic* (abstract workflow), defines the structure of a workflow, reflecting the control and data flow of an application.
- The second level, called *platform specific* (concrete workflow), defines how the workflow should be executed on a Grid infrastructure.

Thanks to the first abstraction level, a workflow structure can be defined with no consideration about the targeted platform on which it will run. Through the second level, a workflow definition can be adapted to a particular Grid platform. Therefore, a workflow definition can be used to inspect and monitor the running and finished workflows regarding these two abstraction levels.

GWorkflowDL provides a graphical notation based on Petri Nets which is an intuitive and simple graphical description, what allows the users the modelling of workflows without having to learn the notation of a specific workflow language.

1.2.3 Orchestration-based Workflow Management Systems

There are many workflow management systems being developed as centralized systems which rely on an orchestrator responsible for the coordination of all the control and data information among services. These systems can be deployed on local desktop computers to help individual users to construct workflows from available services. In the following, we review some of the popular systems: the BPEL Engine for the business domain, and Kepler [86], Taverna [104], Pegasus [54] and MOTEUR [73] for the scientific domain.

1.2.3.1 BPEL Engine

BPEL processes are executed by a so-called BPEL engine. The Web services used in a composition can be distributed among several partners over the network. The whole workflow is described in a single file and executed by a BPEL engine.

Traditionally, a BPEL engine works as a broker for all message exchanges between all the Web services participating in a workflow, in other words, the workflow logic runs on a central server. The engine processes the workflow specification and ensures the correct execution.

A BPEL engine offers a conceptual distinction between abstract processes that describe the global view on the process model, and executable processes that describe workflows and can execute them using a BPEL engine. The BPEL engine exposes every process through a Web service interface, thus allowing this process to be invoked or composed as a simple task in other process definitions.

Nowadays, there are a wide variety of implementations for a BPEL Engine. In fact, most of IT enterprises have developed their own version, thus showing the widespread

adoption of this model, *e.g.*, Apache [4], ActiveBPEL [5], Intalio [6], Oracle [7].

1.2.3.2 Taverna

Taverna [104] is an open-source scientific workflow management system initially targeted for life science, and developed by the myGrid consortium[8]. The primary goal of Taverna is the construction and execution of workflows accessible to scientists. A workflow definition is a linked graph of processors, that represent Web services or other executable components supporting various bioinformatics data analysis and transformation. These workflows are designed using an XML-based workflow language called Scufl (see Section 1.2.2.3), and executed according to a functional programming model. Taverna presents a data-driven coordination model that allows the interaction among different local and remote services provided by external applications.

Initially, Taverna was oriented to the biological domain, however due to its simplicity, it is currently used in other domains such as bioinformatics, chemioinformatics, astronomy, social sciences and music. Therefore, a significant effort was put towards discovering and organizing these Web services into a reusable set of components. Today's Taverna is directly integrated with the myExperiment[9] initiative whose aim is to collect, find, use and share scientific workflows.

Note that, thanks to a plug-in architecture, additional components can be included into Taverna to support secure Web Services and Grid execution.

1.2.3.3 Kepler

Kepler [86] is a Java-based open-source workflow management system developed by the Science Environment for Ecological Knowledge (SEEK) project[10] and the Scientific Data Management (SDM) project at the University of California Berkeley[11]. Kepler is an extension of Ptolemy II [59], a mature system for modelling, simulating and designing concurrent, real-time applications. It also inherits, the support for multiple heterogeneous models of computation from Ptolemy II. These models are captured by the notion of *directors* providing flexible control strategies for the representation of different kinds of systems. Moreover, each computation model is independent of the defined structure for a workflow, which is built by composing a set of components, called *actors*. An actor represents an operation, or data source, that implements a particular functionality for a specific domain. The behavior of an actor can change in function of the execution and communication semantics provided by the adopted director. This characteristic is known as *behavioral polymorphism*.

Like Taverna, Kepler uses a data-driven model based on the actor-oriented composition model. A workflow definition is composed of actors that are linked among them

[4] http://ode.apache.org/
[5] http://www.activevos.com/learn/open-source
[6] http://www.intalio.com/bpm
[7] http://www.oracle.com/technetwork/middleware/bpel/overview/index.html
[8] http://www.mygrid.org.uk/
[9] http://myexperiment.org
[10] http://seek.ecoinformatics.org/
[11] https://sdm.lbl.gov/

through interfaces called *ports*. A port can represent an input, output or mixed parameter. They are connected through channels that are directed from the output port of an actor to the input port of another actor.

In Kepler, the primary goal consists of complex models built hierarchically with the combination of different heterogeneous models with distinct computation models.

1.2.3.4 Pegasus

Pegasus [54] is an open-source workflow system that enables users to execute workflows on large-scale infrastructures such as Grids or Clouds. It provides a framework that maps complex scientific workflows onto available compute resources and executes them in an appropriate order following a workflow specification.

Pegasus supports the abstract workflow definition by allowing users to construct workflows using DAX (see Section 1.2.2.4), without worrying about the underlying execution platform details. The user provides a workflow definition and then artificial intelligence techniques are used for guiding workflow composition, in particular by the moving of data and execution of applications on a distributed, dynamic and heterogeneous set of computational resources.

Pegasus has been identified as the only workflow system covering the whole workflow life cycle, which is a set of linearly connected states of a workflow's life: *creation*, *planning*, *scheduling* and *reuse*.

Pegasus relies on existing Grid infrastructures such as DAGMan [71] and Globus [68] to provide the necessary information for the resource selection. In order to provide a dynamic scheduling of tasks on resources, portions of a workflow can be mapped based on data availability. *Task clustering* is also considered, where a number of small-granularity tasks are executed on the same resource. Pegasus incorporates some static strategies for the resource selection such as random, round-robin and so on. In addition, an adaptive scheduling can be done thanks to the use of the MAPE functional decomposition, allowing a just-in-time scheduling. As illustrated on Figure 1.7, Pegasus is composed of three components:

- **Mapper.** This component transforms an abstract workflow definition into an executable workflow, and finds the appropriate software, data, and computational resources required for its execution. Performance optimization by changing the workflow structure can be also achieved.

- **Central Engine.** This component executes the tasks of a workflow specification based on the data-flow using the DAGMan workflow executor of Condor. DAGMan relies on the resources (compute, storage and network) defined in the workflow to perform the necessary actions.

- **Task Manager.** This component is responsible for the management of workflow tasks, supervising their execution onto the distributed resources.

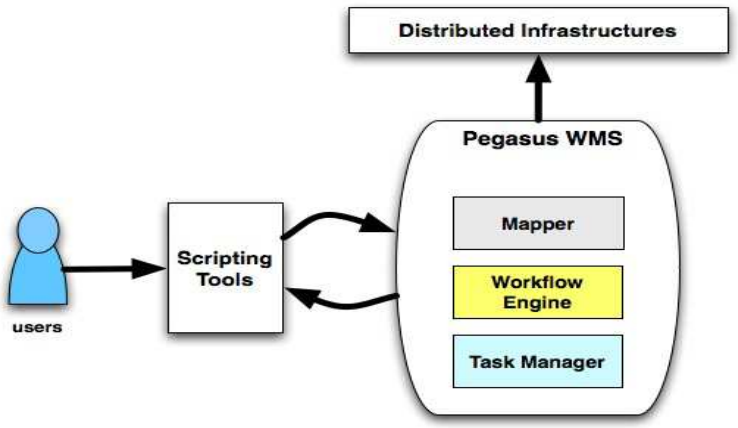

Figure 1.7: Pegasus workflow management system.

Like Taverna, Pegasus is used in a wide variety of domains such as astronomy, earthquakes science and bioinformatics. To facilitate its adoption on additional domains, Pegasus includes a key feature which is the support for the execution of scripts composing components programs rather than Web services.

1.2.3.5 Moteur

MOTEUR [73] is a centralized workflow engine developed by the Modalis Team [12] at the University of Nice Sophia Antipolis. The main feature of its execution model enables data, workflow and services parallelism when processing a workflow definition. This workflow engine allows to define data-flows through the composition of services and executable components. Based on the data-intensive nature of scientific workflows, MOTEUR supports some data composition strategies such as *one-to-one* and *all-to-all*. For instance, the *all-to-all* strategy corresponds to the scenario in which all the inputs in one dataset have to be processed with all inputs in the other dataset.

The authors adopted the Scufl language for the workflow definition (see Section 1.2.2.3). As a consequence, this engine presents some expressiveness limitations when modelling complex control structures, as we explained in Section 1.2.2.3. More recently, a new workflow language was developed in the context of the GWENDIA project [13] to run scientific workflows in MOTEUR, called GWENDIA [6]. GWENDIA is a data-driven workflow language that provides transparent access to Grid infrastructures for coherently and efficiently resource usage. This language can convert Scufl specifications into GWENDIA specifications. However, GWENDIA is still based on Scufl, so it also includes coordination links to construct control structures, offering a poor expressiveness when modelling complex workflow patterns.

Furthermore, MOTEUR provides supports for processing of large and dynamic datasets, and is currently used to run workflows written in the Scufl language on environments like *The Grids for e-SciencE (EGEE) [60]* or *Grid'5000 [8]* .

[12] http://modalis.i3s.unice.fr
[13] http://gwendia.i3s.unice.fr/doku.php?id=gwendia

1.3 Choreography

With the blooming of Web technologies, more and more real computations are established through the composition of several computation units. These units have to work correctly, but also ensure a successful collaboration with each other in order to achieve a shared goal. These computation units can represent independent entities or organizations locally distributed over a network.

When modelling these scenarios, orchestration approaches suffers from the following drawbacks:

- The ever growing complexity of the interactions lead to some scalability and interoperability issues.
- The correctness verification of the interactions is even harder from the perspective of one participant.
- In real-world scenarios, organizations are often unwilling to delegate control of their business processes to their integration partners.

Choreography offers a way to clearly define and agree on the rules of participation within a service collaboration, as mentioned in [12]. Each participant (computation unit) may then implement its portion of the choreography as determined by the global perspective.

The intent behind choreography is to make it easy to check the conformance of each participant to the expression of the global system it takes part in [12, 91].

Choreography, although an established concept, has less literature and real implementations than orchestration. In practice, the design processes and execution infrastructure for service choreography models are inherently more complex than for service orchestration: decentralized control raises a new set of challenges which are the result of message passing between distributed asynchronous-concurrent processes. However, although more complex, there are some benefits by adopting choreography models [29]:

- **Abstraction level:** As we mentioned before, to design a choreography system, engineers need to have a global view of how the services interact with each other. Hence, the key lies not on building individual services but rather on how sets of services collaborate together, by identifying groups of services and analyzing and understanding their interactions.
- **Modularity:** In a choreography model, each participant is an independent entity that plays a pre-determined and pre-agreed role regardless any central orchestrator.
- **Scalability:** Centralized coordination through a single orchestration engine is a valid solution for scenarios found in the business domain, where relatively small quantities of intermediate data (when output from one service invocation is used as input to another service) and computation are processed and transferred between services. However, when processing computation and data intensive workflows, there is an increment on unnecessary data transfer, wasted bandwidth and workload for this central engine, what obviously decreases the speedup of the system when processing this type of workflows.

To sum up, a choreography model defines the set of message exchanges corresponding to interrelated service interactions.

1.3.1 Choreography Languages

In this section, we introduce the main attempts at building languages for the definition of service choreographies.

1.3.1.1 WS-CDL

The Web Service Choreography Description Language (WS-CDL) [12, 29] is an XML-based language for describing the interactions among multiple participants in a service composition. WS-CDL defines the control and data flow among services from a global perspective.

In WS-CDL, each service participant has a role that represents its behavior during the collaboration, *i.e.*, *Provider* or *Consumer*. As well, a participant can also play different roles depending on the interaction. Like BPEL, WS-CDL has a construct named *activity* that identifies the tasks to be executed. There are three types of activities: *basic*, *workunit* and *structural*. The *basic* activity specifies the interaction between two participants. The *workunit* activity defines loops and conditional constructions, and the *structural* activity builds control structures such as sequences and parallel splits. WS-CDL also supports mechanism for exception handling during the execution, as well as the possibility to define sub-choreographies inside of an existing one forming a tree-like structure.

WS-CDL takes its formal roots in π-*calculus* [70], a formal basis for the description of concurrent process and dynamic interconnection scenarios. However, it has been a W3C Candidate Recommendation since November 2005. There are several reasons to explain this status [12]. The lack of execution details could be considered as the main reason. Others drawbacks such as no multipart support and tightly bound to WSDL interfaces are devised in [12].

1.3.1.2 BPEL4Chor

BPEL4Chor [52, 53] is an extension of BPEL language for modelling choreographies. While BPEL is a language for describing orchestration models, BPEL4Chor allows to interconnect the business process from multiple patterns to achieve a common objective.

BPEL4Chor distinguishes between three artifacts types when defining a specification:

- *Participant behavior descriptions (PBD).* It defines the control and data dependencies between the participants, in particular, between activities (tasks).
- *Participant topology (PT).* It provides a structural view of the composition by specifying the participants and their interconnection using message links. This artifact is used in conjunction with PBD artifacts to obtain an abstract choreography specification.
- *Participant groundings (PG).* It defines the configuration details such as data formats and port types from WSDL definitions.

BPEL4Chor encourages reuse by only providing a specific Web service mapping in the participant grounding. Unlike, in WS-CDL, a composition with an unknown numbers of participants can be modeled.

Recently, in [52], authors proposed to use both BPMN (Business Process Modelling Notation) [112] and BPEL4Chor for processing business collaboration among multiple partners. BPMN provides a friendly graphical notation for modelling business processes, while BPEL4Chor transforms these definitions into executable business processes.

1.3.1.3 Let's Dance

Let's Dance [51, 147] is a language for describing the collaboration among services from a global and local perspective. Thanks to both perspectives, Let's Dance supports the modelling of interactions between a set of services from one observer's viewpoint, and the modelling of the interactions in which a particular service is directly involved. Using this language, a choreography consists of a set of interrelated interactions specified by message exchanges among services.

Like WS-CDL, each participant can have different roles in a service interaction. In order to design a service interaction, one of three following constructs has to be selected:

1. *Precedes.* This construct denotes that the target interaction can only be triggered once the *source interaction* has completed.

2. *Inhibits.* This construct denotes that the target interaction can no longer occur once the source interaction has completed.

3. *Weak precedes.* This construct denotes that the target interaction can only be triggered once *source interaction* has completed or it has been inhibited.

Let's Dance also supports the construction of the service interaction patterns introduced in [76]. This allows to model a wide range of collaborations by using patterns such as send/receive, one-to-many send/receive and multi-responses, among others.

1.3.1.4 MAP

MultiAgent Protocol (MAP) [30] is an XML-based executable language that expresses the collaboration among multiple patterns. Considering a set of peers over a network, a choreography definition does not have to be installed in advance at design time on each peer participating in the interaction. Instead, peers can be dynamically configured to collaborate with others for processing a choreography specification. Therefore, MAP provides a peer-independent and flexible solution for the execution of choreographies, in which the definitions can be uploaded to a set of peers at runtime.

To design a choreography model, an engineer must first identify different roles, and then extract the implicit protocol defined by the interactions among roles. Each peer is identified by a unique name and a role. So, a protocol can be thought of as a bounded space in which a set of peers collaborate to achieve a common goal.

MAP offers a mechanism for both asynchronous and synchronous service invocation and also provides support for the construction of control structures such as sequences, exclusive-choices, parallel splits and loops among others.

1.3.2 Choreography-based Workflow Management Systems

Most of actual implementations of the models use proprietary languages for the execution of choreographies. However, other approaches [52, 69], use more standardized languages, like BPMN or WS-CDL, to obtain abstract choreography descriptions that will be finally turned into BPEL definitions for the execution. In the following, we detail two of these implementations: Maestro [51] and MagentA [30].

1.3.2.1 Maestro

Maestro [51] is an implementation of the Let's Dance language for the analysis and simulation of global and local service interaction models. As a part of a modular architecture, Maestro is built upon the Maestro visual language framework developed by SAP [14]. This framework provides an editor environment for developing BPMN, BPEL and SAM (Status-and-Action Management) specifications, what allows to design more abstract choreography models.

The choreography model is used as an input argument for the analysis and the simulation components in Maestro. The analysis component includes a mechanism which is able to detect any semantical error such as unreachable interaction, and also allows to obtain how many times an interaction can occur during an execution. This mechanism avoids the detection of future communication problems, as well as determines what desirable characteristics should offer a communication channel. The simulation engine allows to execute instances of a specification to verify the correctness of a model.

1.3.2.2 MagentA

MagentA [15] is an open-source framework for the enactment of choreographies defined using MAP language (see Section 1.3.1.4). This framework was implemented using a combination of Java, XML and Web technologies.

As mentioned before in Section 1.3.1.4, MAP processes a choreography definition across a set of peers in a network by using message-passing to communicate. In MagentA, each peer is a dynamic and lightweight piece of middleware that serves as a proxy to a service or a group of services.

In addition, peers are exposed via WSDL descriptors, what increases the flexibility of the system, *i.e.*, additional peers can be included to improve the performance during data transfers between services. Similarly, MagentA provides a registry lookup service to locate and configure the peers at runtime. The physical location of each peer will be spliced in the choreography specification before it is disseminated to the participants for the execution.

MagentA has been successfully applied to execute real choreographies on a variety of e-Science projects [30].

[14] http://www.sap.com/france/index.epx
[15] homepages.inf.ed.ac.uk/cdw/

1.4 Conclusion

In this chapter, we presented the service oriented computing as a paradigm emerged by the composition of services which collaborate among them to achieve a common goal. The service composition is not a trivial endeavor. Two main composition methodologies exist for the definition of the service interactions: orchestration and choreography. Both methodologies define the collaborations on the use of *loosely coupled* services from opposite viewpoints. These different viewpoints have caused passionate debates between designers to decide which form of composition to use, as mentioned in Section 1.1.2. Nowadays, even though the orchestration has gained a lot of adoption in the software industry, this discussion remains an open issue when considering the emerging large-scale distributed systems. While the orchestration, it is considered as the *defacto* standard for modelling workflows, which are currently managed in a centralized way. Designers consider the choreography as an appropriate technique for modelling abstract workflows, which are managed in a decentralized way. However, the existing choreography models present a tight coupling interaction mechanism, what limits its adoption for the management of workflows.

As a consequence, we explored the advantages and disadvantages of the existing architectures for service composition suggesting a promising future to those that can incorporate loose coupling and decentralization. Today, the majority of workflow management systems are basically managed in a centralized manner, what may provide a poor scalability and reliability for the execution of data and computation intensive workflows. Similarly, current workflow languages exhibit several limitations regarding dynamic adaptation, due to their low level of abstraction and static nature (*i.e.*, explicit parallelism and static binding).

Therefore, there is a demand for more flexible coordination models to process the next generation of service composition systems, which should run at large-scale and provide abstractions for decentralization and dynamic behavior.

Chapter 2

Flexible Models for Service Coordination

As we mentioned at the end of the previous chapter, the existing languages and workflow management systems are basically static, tightly coupled and management-centric. The languages define workflow specifications to be processed by a central engine. In traditional choreography models, services interact directly offering a tight coupling interaction mechanism. The services collaborate through message-passing by transferring the workflow definition across them, as proposed in [63, 146].

The volatility of services requires dynamic coordination models that provide the ability to quickly adapt to changes, as providers do not continuously supply their services. Languages like BPEL or WS-CDL exhibit some limitations with respect to dynamic adaptation. Even though some of them support dynamic service binding, they lack adaptation mechanism regarding changes in the composition structure.

Based on this constation, several authors proposed to extend these languages using aspect-oriented programming (AOP) for supporting dynamic adaptation at runtime, as suggested in [46, 49]. However, there are some negative points around of AOP such as the limited modularization and reusability, difficulty of understanding and debugging. Therefore, these approaches were not widely adopted since they increase the complexity of the service composition systems. To solve the tight coupling of these coordination mechanisms, the partitioning of workflow definitions seems to be a good solution to reduce the workload of central engines. Nevertheless, partitioning is a complex task when it has to be done statically at design time. There is a lack of generic and simple approaches to decentralize the execution of a service composition [11, 63, 118].

These drawbacks encouraged the development of more flexible (dynamic and loosely coupled) models for service coordination, in contrast to the pre-determined and pre-specified properties of the classical service composition solutions.

This chapter presents different approaches injecting loose coupling and dynamicity in service coordination, relying on rule-based systems and tuplespace-based architectures. Then, we discuss the foundations and languages of the chemical programming model, and its adequacy to nicely express coordination. This part is detailed in Section 2.1. Second, loosely coupled coordination mechanisms that enables asynchronous interactions among services, based on a *tuplespace*, are presented in Section 2.2. Third, the principles and languagues of chemistry-inspired models, built on top of both rule-based systems and

tuplespaces, as well as the existing literature about their adequacy to the expression of service coordination, are detailed in Section 2.3. Finally, Section 2.4 summarizes the ideas and concepts exposed in this chapter.

2.1 Rule-based Models

The concept of rewrite rules is presented in different fields of computer science, both theoretical and practical, *e.g.*, to implement program transformations, to define constraint-based algorithms, etc... Rule-based programming provides a common framework in which computation can be seen as a set of logical statements about how a system operates.

In [47, 141, 143], the authors consider that rules can be used in the context of service composition to determine how the composition should be structured and processed, how the services and their providers should be selected, and how run-time service binding should be conducted. Rule-based approaches are managed and guided by rules to support service composition and dynamic binding of services. Furthermore, these approaches provide a high level of abstraction for the modelling of the service interactions, as rules allow to define the collaborations without having to interact with the individual services.

The majority of rule-based systems use a type of rules called *Event-Condition-Action*. Event-Condition-Action (ECA) rule was originally suggested as a formalism for active database capabilities [90]. An ECA rule consists of a triggering event, a list of conditions, a performed action and an optional postcondition, which formalizes the state change after the execution of the action. Rules define flow, constraints and control the behavior of a system by reacting to events, what allows the construction of *event-driven* applications.

In the work [143], the authors have developed an ECA rule-based workflow management system for Web service composition using rules. This system uses the high level of abstraction to easily construct workflows models using Web services, without having to interact with individual services composed. In [82], the authors propose semantic and a rule based event-driven system for automation of business processes. Semantic provides knowledge-vocabulary of domain and ECA rules are designed to generate the composition schema automatically and dynamically according to the events.

On the other hand, in [47], the authors proposed a flexible composition framework for Web services development, and a rule-based language for service integration. This framework exploits the pattern matching mechanism of rewriting systems for XML processing, thus allowing rule-based services to interact with other RPC-style services.

Therefore, as mentioned in these approaches, rule-based models are able to design with a high level of abstraction service interactions supporting dynamic adaptation and service composition life cycle management.

2.2 Tuplespace-based Models

There is a need to specify the interactions among services via the definition of the control and data flow of their interactions. Accordingly, the concept of tuplespace arose as a suitable mechanism for modelling asynchronous interactions among services. Conceptually, a tuplespace is similar to an associative (possibly distributed) shared space, that allows loosely coupled interactions between a set of processes, which can be *producers* or *consumers* of information, Tuplespace supports advanced information exchange patterns

such as *one-to-one* and *one-to-many* interactions. This shared space is usually thought of as a 'blackboard' that facilitates the coordination of distributed elements.

Therefore, the main advantage of a tuplespace for the composition of services is loose coupling, since the interactions do not occur directly among Web services, but are now mediated by this coordination space. In the tuplepsace, all the data exchanged among services are stored and accessed asynchronously [85]. This coordination model improves the reusability by simplifying the development and management of Web services. Applications do not need to be implemented based on the interfaces of the services participating in a composition.

In the following, we introduce the first coordination model founded on the tuplespace principles called *Linda* [130]. The Linda tuplespace provides a simple and elegant way of separating communication from computation concerns. Finally, we explore the more relevant tuplespace-based approaches for service coordination.

2.2.1 Preliminaries

Originally, the tuplespace was conceived in Linda to act as a distributed and shared-associative memory platform on which to build applications. Thus, a tuplespace can be seen as a shared space that provides primitives for storing and retrieving ordered data objects under the shape of *tuples*. There are two types of tuples: *entries* and *templates*. An entry consists of a tuple in which all the fields have a defined value. A template, denoted by $\bar{t}$, represents a tuple with one or more undefined fields, and its structure defines the matching pattern to access to tuples in the tuplespace.

As illustrated in Figure 2.1, the original Linda model presents three primitives to be performed on the tuples and the tuplespace: *out(t)*, *in($\bar{t}$)* and *read($\bar{t}$)*. The primitive *out(t)* is used to insert a tuple t in the tuplespace; *read($\bar{t}$)* is used to read one tuple from the tuplespace without withdrawing it, as long as there is a tuple that matches the template $\bar{t}$. A tuple can be read and withdrawn using the *in($\bar{t}$)* operation. The *in* and *read* are blocking operations respectively.

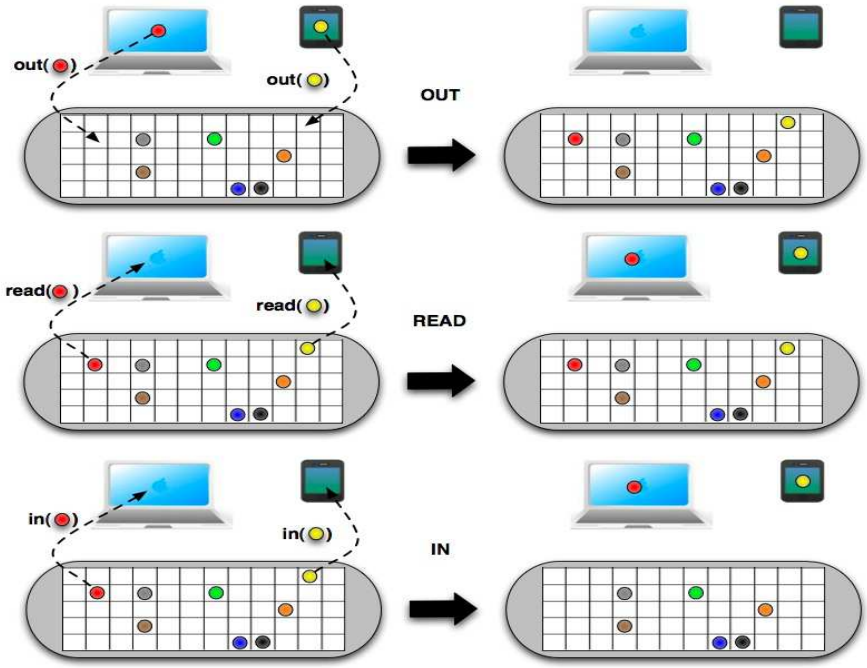

Figure 2.1: Tuplespace operations.

There has been a wide diversity of software projects based on the tuplespace model, such as XMLSpaces.NET [1], TSpace [2] and JavaSpaces [3]. In particular, JavaSpaces implements a tuplespace mechanism for coordination, dynamic communication, and sharing of Java objects between resources in common client-server networks. Thus, this mechanism allows participants to exchange tasks, requests, and information (resources or objects) by acting as a virtual shared space between the providers and consumers in a distributed application. JavaSpaces is an interface which can be instantiated with the Java Jini technology [4].

2.2.2 Existing Approaches

Today, there is a growing interest in using tuplespace-based coordination models for service composition, or to simply coordinate the information exchanged between different parties [31, 40, 103, 138]. In the following, we focus on Web service composition through tuplespace.

The main attractivenesses of this model are the following:

- **Asynchronous interaction:** the information can be exchanged between services instances even if they are already deployed or under deployment. Thereby, services

[1] http://www.ag-nbi.de/research/xmlspaces.net/
[2] http://www.almaden.ibm.com/cs/TSpaces/
[3] http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/
[4] http://java.net/projects/jini/

instances do not necessarily have to exist during the same period of time to exchange the information.

- **Storing and searching facilities:** a tuplespace provides some primitives to collect and post tuples when needed.

- **Advanced message exchange patterns:** A tuplespace supports notifications to service instances, which indicate, using templates, the kind of data they are interested in. Thus, the matching pattern is based on the relevant data for a service instance instead of being based on its location or data provider.

- **Loosely coupled interaction:** Traditionally, service-based applications suffer from tight coupling interactions between services. These collaborations are programmed according to the interface of the services (if the interface change, the application may fail). With tuplespace-based, services interface do not need to be completely known.

Next, we make a critical review of the most relevant approaches using a tuplespace as a coordination mechanism among services (like an orchestrator). Note that, other approaches using a tuplespace as a coordination mechanism among workflow engines for a decentralized execution will be discussed in Section 3.4 of Chapter 3.

Thus, this coordination model was also proposed to avoid tight coupling interactions among services using the workflow engines of languages such as BPML [17] and BPEL [38].

In [40, 103], the authors present a loosely coupled and private tuplespace as an indirect communication mechanism to support Web based collaboration and XML data. Both approaches allow to store XML documents in a tuple field and retrieve them. Furthermore, the authors included a mechanism to control the access between public and private tuples. In the same vein, but more recently, a tuplespace called *xSpace* was presented in [31]. There, the authors focused on the efficiencies that can be wrung out of a pure XML tuplespace rather than hybrid tuplespaces like those mentioned above.

As illustrated on Figure 2.2, a tuplespace supports the insertion of XML documents, in particular, SOAP messages for the service coordination. Consider a service collaboration between '*service*1' and '*service*2', '*service*1' may communicate with '*service*2' to continue the execution. Then, '*service*1' inserts a tuple, a SOAP request message, in the tuplespace referred to '*service*2'. Next, '*service*2' withdraws this SOAP request and inserts the result of its operation as a tuple, a SOAP response message. Finally, '*service*2' withdraws the tuple representing the SOAP response and continues the execution. This example easily illustrates how a tuplespace, with support for XML, could be used for the service coordination.

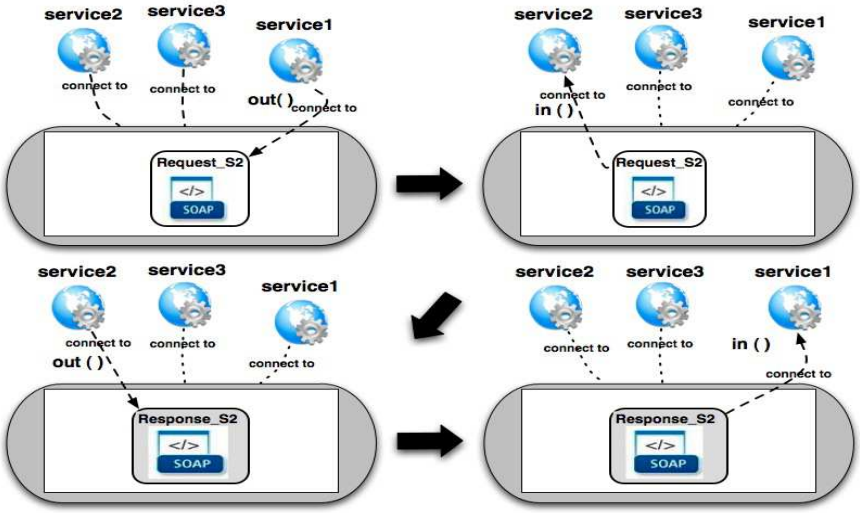

Figure 2.2: Service coordination using an XML tuplespace.

Therefore, the tuplespace provides to service providers and consumers asynchronous interaction and three decoupled dimensions: destination, space, and time. The service providers of data can publish it at any point in time (time), independent of its internal storage (space), independent of the knowledge about potential consumers (destination). The coordination model based on the tuplespace facilitates the dynamic binding and service composition. However, the use of this model for Web service composition also present some disadvantages:

- The tuplespace normally has to support XML data in order to facilitate the interoperability among Web services, as pointed out in [31, 40].
- These systems do not normally offer any mechanism for the control of information being exchanged during a coordination process, as detailed in [40, 103], .
- There is a lack of autonomic behavior to solve possible conflicts into the system, *i.e.*, the tuplespace is not aware of any problem during the interactions.
- There are no primitives describing complex interactions among services, such as complex synchronization patterns.

As a result, we considered the disadvantages and advantages of this coordination model in order to design our system. In such a way, our control and data repository system has some similarities with these tuplespace-based models in the sense that loosely coupled and asynchronous interactions are also supported.

2.3 Chemical Metaphor-based Models

There is a growing literature on the use of the chemical metaphor to design new computation models [24, 33, 105, 110]. The chemical programming model is an elegant

implicitly parallel programming model, initially proposed to simply write highly parallel programs. The metaphor is as follows. A chemical program can be seen as a chemical solution where data is represented by "floating" molecules and computation by chemical reactions. Molecules interact freely according to Brownian motion and react according to some conditions. When some molecules match and fulfill a reaction condition, they are consumed and replaced by the result of the reaction (producing new molecules). This process goes on until a stable state where no more reactions are possible is reached, referred to as *inertia*.

In the following of this chapter, we first introduce the chemical programming models and their benefits for service composition.

2.3.1 Gamma

The Gamma formalism [24], proposed in 1986, is the pioneer programming model inspired by the chemical metaphor. Using Gamma, a program consists of a reaction (condition) that consumes some molecules and produces new ones. The unique data structure in Gamma is the multiset (*i.e.*, a set of possible multiple molecules) on which the reactions are applied. Thus, the execution consists on the consumption of molecules (reactants) satisfying the reaction condition by new molecules produced by this reaction. The end of the execution is determined when the *inertia* state is reached. For example, a Gamma program calculating the maximum element of a set of integers will be defined as following:

$$\text{max} = \textbf{replace}\ \text{x, y}\ \textbf{by}\ \text{y}\ \textbf{if}\ \text{x} \leq \text{y}$$

The condition $x \leq y$ represents the reaction condition to be satisfied by the molecules x and y. If the condition is satisfied, the molecule x is removed, leaving the molecule y in the multiset. In Gamma, the execution of a program is non-deterministic. A non-deterministic program defines an algorithm with one or more choice points where multiple different continuations are possible, without any specification of which one will be taken. Molecules autonomously react according to the rules until inertia state is reached. They are consumed without any pre-defined order. Gamma is an implicit parallel language since several reactions can simultaneously react, as long as there are no molecules taking part to several reactions at the same time and their reaction conditions are satisfied. Moreover, it also supports the atomic capture of molecules. The reaction takes all the reactants atomically, otherwise the reaction do not take places. In other words, once all the reactants are present into the solution, none of them can participate in more than one reaction at the same time.

Let us consider a set of integers (9, 6, 4, 2) for our *max* example, the molecules 6 and 9 could react together although the molecule 9 could also react with the 2 or with 4 following a different order of capture. Indeed, the 9 and 6 molecules can also react in parallel with the 4 and 2 to obtain the maximum value 9.

Gamma is a simple and high level programming language for parallel applications in comparison with the existing parallel programming models. This language is able to express parallel and non-deterministic programs without an explicit sequentiality.

2.3.2 Similar Models

The pioneer chemical model proposed in Gamma has inspired a variety of authors to design other chemical models such as the CHAM (CHemical Abstract Machine) [33] and P-systems [105].

Chemical Abstract Machine. CHAM was proposed by Berry and Boudol for asynchronous and concurrent computation without explicit control. This non-deterministic model was based on Gamma and added some new features such as *membranes* and *airlocks* mechanism. The need of these new features emerged from the description of distributed systems using models such as Algebraic Process Calculi [95]. Membranes are used to contain molecules and other membranes, creating a tree-like structure of subsolutions. Airlock mechanism was introduced to describe communications between membranes. This mechanism enables one molecule to be accessed from outside the membrane and this to take part in a reaction in the root membrane.

Like Gamma, CHAM presents the state of inertia and also reactions are n-shot rewriting rules. A n-shot rule is a kind of reaction that never disappears of its enclosing solution after reacting.

P-systems. This approach is a computational model inspired from biology and based on a structure composed by membranes. Membranes could be considered as cells for analogy with biology or chemical solutions with chemistry. Thus, a membrane is a multiset containing other membranes and data, where data are the molecules "floating" into these membranes. This model consists in nested membranes, which can be seen as a tree-like structure, in which molecules react in. Molecules can cross and move through the membranes independently of its level into the tree structure, what enables the modelling of interactions (communications) among membranes.

A set of partially ordered rewrite rules is associated to each membrane describing possible reactions and interactions among them. The execution of a rule can consume molecules but also membranes. For instance, if a reaction dissolves a membrane, then the molecules from that membrane will be associated to its new enclosing membrane.

2.3.3 Higher-Order Chemical Programming

In order to satisfy the requirement of dynamicity in the emerged distributed systems, the chemical programming models needed to offer a dynamic behavior which enabled to adapt these systems to the changes in the environment. Reaction rules cannot be modified dynamically in Gamma. Thus, as a consequence, *Banâtre, Fradet, Radenac* decided to include the higher-order as a novel feature for the new generation of chemical models thus providing such demanded dynamicity.

The higher-order chemical programming language allows the programmer to define reaction rules which can consume and produce reaction rules. Next, we introduce two of the existing higher-order chemical models and present how these new extensions can satisfy the requirements of those distributed systems.

M	:=	x,y, ... ϵ V	*; variables*
	|	$\gamma\langle$x$\rangle$.M	*; γ-abstraction*
	|	M_1, M_2	*; composition of molecules*
	|	$\langle$M$\rangle$	*; solution*

Table 2.1: Syntax of γ-calculus.

2.3.3.1 γ-Calculus

The γ-calculus [21, 110] is a higher-order chemical model where rules are also considered as molecules – "first class citizens" –, that float in the multiset. Like in Gamma, the unique data structure of the γ-calculus is the multiset. Chemical solutions contain these molecules and work as biological membranes isolating them from others. Hence, molecules into a solution cannot react with others outside of this solution, however they can be extracted or added from its enclosing solution using reactions.

The syntax of γ-calculus is illustrated on Table 2.1. A γ-abstraction specifies a higher-order rewrite rule. The reaction concept is represented by a single rewrite rule of the form:

$$\gamma\text{-abstraction} \equiv (\gamma \langle \text{ P } \rangle.\text{M})$$

When a γ-abstraction finds a molecule N that matches the pattern P, it reacts and is replaced by the molecule M. The execution consists of γ-reactions until the inertia state is reached for the solution representing a chemical program. Note that some γ-abstractions disappear from its enclosing solution after reacting. This kind of rules are called *one-shot*. In γ-calculus, there are two types of rules: *n-shot* and *one-shot*, while there is only *n-shot* rules in Gamma.

As we mentioned before, reaction rules are also considered as molecules in γ-calculus, reaction rules can then consume and produce other reaction rules. To do that, we first have to consider an important restriction that permits to rewrite reactions following a certain order, — A molecule can be extracted from its enclosing solution only when it has reached an inert state —. Without this limitation, the contents of a solution could be extracted in any state and the solution construct would lose its purpose.

The γ-calculus can be seen as a formal and minimal basis for chemical languages, as well as the λ-calculus is the formal basis of functional programming languages. In fact, the λ-calculus can be encoded withing γ-calculus. Constructions such as logical, arithmetics, integers, tuples, recursion, booleans and conditionals can be defined using both paradigms. Nevertheless, the γ-calculus is more expressive than λ-calculus since non-deterministic programs can also be expressed.

Despite the high expressiveness of the γ-calculus, it lacks two fundamental features in comparison with other chemical models such as Gamma, CHAM and P-systems :

- **Reaction condition.** While in Gamma, reactions take place if a condition is satisfied , inertia and termination are described syntactically giving to the conditional reactions a semantic nature in γ-calculus. In other words, there is not any syntax

in γ-calculus to define a reaction condition. However, the termination and inertia are inherently defined through the reaction rule syntax (left hand side).

- **Atomic capture.** In Gamma, any fixed number of elements can be consumed by a reaction. While the γ-abstraction reacts with one element at a time, a n-ary reaction takes atomically n elements which cannot be consumed by any other reaction at the same time.

2.3.3.2 Higher-Order Chemical Language

In this section, we present a language based on the γ-calculus but including the needed features to be useful as a programming language. This language improves some syntactic constructions of γ-calculus such as the condition reaction and the atomic capture of molecules as an extension like those provided by Gamma in Section 2.3.1. Based on that, *Banâtre, Fradet, Radenac* designed a new chemical programming language called Higher-Order Chemical Language (HOCL) [110]. HOCL, although based on the γ-calculus, represents an extension of Gamma that enables rules to be applied on others rules (programs modifying other programs).

Two of the more relevant properties of HOCL are explained in the following:

- **Condition Reaction.** The reaction rules enable to express the atomic capture of molecules depending of the type (dynamically checked) and structure (tuples, sub-solutions). Similarly, conditions can be also defined as pre-conditions to trigger a reaction. HOCL allows the programmers to use two types of rules: *one-shot* rule based on the proposed by γ-calculus; and the *n-shot* rule based on the proposed by Gamma, that never disappear after reacting.
- **Atomic capture.** A reaction rule takes all its elements (reactants) atomically. Either all the required elements are present or no reaction takes place. If all the required elements are present, none of them can be consumed by another reaction at the same time.

Syntax of the Language

Next, we briefly present the different elements of HOCL syntax, as illustrated on Table 2.2

Reaction rules. Reactions are described based on γ-reactions, and executed locally into the enclosing solution. A rule only reacts when the molecules match the pattern associated or when a condition reaction is satisfied:

$$\begin{gathered} \text{x match M} \equiv \{\text{x} \Rightarrow \text{M}\} \\ \langle\ \text{P}\ \rangle \text{ match } \langle\ \text{M}\ \rangle \equiv \text{P match M if } \langle\ \text{M}\ \rangle \text{ is inert} \\ \text{P}_1, \text{P}_2 \text{ match } \text{M}_1, \text{M}_2 \equiv \text{P}_1 \text{ match } \text{M}_1 \oplus \text{P}_2 \text{ match } \text{M}_2 \\ \text{P}_1{:}\text{P}_2 \text{ match } \text{M}_1{:}\text{M}_2 \equiv \text{P}_1 \text{ match } \text{M}_1 \oplus \text{P}_2 \text{ match } \text{M}_2 \end{gathered}$$

The variable x represents any molecule M, so if x *match* M the variable x is associated to the molecule M. The solution $\langle \text{P} \rangle$ represents any inert solution $\langle \text{M} \rangle$ if the content of

solution M matches the pattern P, the variables of P will be mapped to $\langle M \rangle$ sub-solution. This mechanism is similarly used for the other patterns: P_1, P_2 and P_1:P_2. Note that the $\oplus$ operator is used to merge two sub-solutions to a unique solution. In the following, we introduce the two types of rules: *one-shot* and *n-shot*.

HOCL adopts the non-recursive abstraction from γ-calculus for *one-shot* rules. Thus, an HOCL *one-shot* rule in comparison with γ-calculus *one-shot* rule has the next form:

$$\textbf{one } \mathrm{P} \textbf{ by } \mathrm{M} \textbf{ if } \mathrm{C} \equiv \gamma(\mathrm{P})[\mathrm{C}].\mathrm{M}$$

Similarly, HOCL adopts the Gamma notation for the *n-shot* rules which are equivalent to the recursive abstractions from γ-calculus in the following way:

$$\textbf{replace } \mathrm{P} \textbf{ by } \mathrm{M} \textbf{ if } \mathrm{C} \equiv (\textbf{ let rec } \mathrm{r} = \textbf{one } \mathrm{P} \textbf{ by } (\mathrm{M},\mathrm{r}) \textbf{ if } \mathrm{C} \textbf{ in } \mathrm{r})$$

According to the chemical metaphor, the *n-shot* rules are known as *catalyzers* since they never disappear after reacting.

Expressions. The expressions are statically typed using standard types, as shown on Table 2.2. HOCL supports the definition of integers, booleans, string constants and associated operations. The solution may contain molecules of different types which serve to select values through patterns. The associated pattern-matching rule is:

$$x{::}T \textit{ match } N \equiv \{\ x \Rightarrow N\ \} \textit{ if Type } (N) \leq T$$

We make use of type inference to avoid type annotations in patterns. For instance, we may write ***one*** *x* ***by*** *x + 1* ***if*** *V* instead of ***one*** *x::Int* ***by*** *x + 1* ***if*** *V* since the type of x can be statically inferred.

A molecule only reacts when it has the appropriate type. For instance, the next solution is inert due to this rule only reacts with integers, not with pairs (X:3).

$$\langle\ \mathrm{X}{:}3, \textbf{ replace } \mathrm{x} \textbf{ by } \mathrm{x} + 1\ \rangle$$

Pairs. Pairs or tuples are denoted by A1:A2, is very standard. Note that the elements of a pair are atoms. Pairs of solutions allow to isolate compound molecules from each other. The associated pattern-matching rule for pairs is:

$$(\mathrm{P}_1{:}\mathrm{P}_2) \text{ match } (\mathrm{N}_1{:}\mathrm{N}_2) \equiv \phi_1 \oplus \phi_2 \text{ if } \mathrm{P}_1 \text{ match } \mathrm{N}_1 = \phi_1 \wedge \mathrm{P}_2 \text{ match } \mathrm{N}_2 = \phi_2$$

We can also define tuples of atoms:

$$\mathrm{A}_1{:}\mathrm{A}_2{:}\ldots{:}\mathrm{A}_n \equiv \mathrm{A}_1{:}(\mathrm{A}_2{:}(\ldots{:}(\mathrm{A}_n)\ldots))$$

Empty solutions. The notion of empty solution in HOCL comes from the pattern ω which can match any molecules even the empty solution (introduced below). This pattern is very convenient to extract molecules from a solution. For example, the following reaction extracts 1's from the content of the solution defined as an input element for this reaction.

$$\mathrm{rmunit} = \textbf{replace } \langle x, \omega \rangle \textbf{ by } \langle \omega \rangle \textbf{ if } x = 1$$

		Solutions	
S	:=	$\langle \mathrm{M} \rangle$	*; solution*
	|	$\langle\rangle$	*; empty solution*
		Molecules	
M	:=	x	*; variable*
	|	$\mathrm{M}_1, \mathrm{M}_2$	*; composition of molecules*
	|	A	*; atom*
		Atoms	
A	:=	x	*; variable*
	|	[name=]**replace-one** P **by** M **if** V	*; one-shot rule*
	|	[name=]**replace** P **by** M **if** V	*; n-shot rule*
	|	S	*; solution*
	|	V	*; basic value*
	|	$(\mathrm{A}_1{:}\mathrm{A}_2)$	*; pairs*
		Basic values	
V	:=	x | 0 | 1 | ...| $\mathrm{V}_1{+}\mathrm{V}_2$ | $\text{-}\mathrm{V}_1$ | ...	*; integers, booleans, strings*
	|	**true** | **false** | $\mathrm{V}_1 \wedge \mathrm{V}_2$ | ...	
	|	$\mathrm{V}_1 = \mathrm{V}_2$ | $\mathrm{V}_1 \leq \mathrm{V}_2$ | ...	
	|	"string" | V_1 @ V_2 | ...	
		Patterns	
P	:=	x::T	*; molecule with type T*
	|	ω	*; any molecule*
	|	name = x	*; naming a reaction*
	|	$\langle \mathrm{P} \rangle$	*; inert solution*
	|	$(\mathrm{P}_1{:}\mathrm{P}_2)$	*; pair*
	|	P_1,P_2	*; composition of molecules*
		Basic types	
B	:=	Int | Boolean | String	

Table 2.2: HOCL syntax.

The pattern ω matches the rest of the solution which is returned as result. If the solution contains only a 1 then ω matches the empty molecule and the empty solution is returned:

$$\text{rmunit}, \langle\ 2,1,3\ \rangle \rightarrow \langle\ 2,3\ \rangle$$
$$\text{rmunit}, \langle\ 1\ \rangle \rightarrow \langle\ \rangle$$

The rule for pattern-matching ω has the next form:

$$\omega \text{ match M} \equiv \{\ \omega \Rightarrow \text{M}\ \}$$

The empty molecule is introduced through rules with a pattern of the form (P,ω):

$$\text{(P,}\omega\text{) match M} \equiv \{\ \text{(P match M}_1\text{)} \oplus (\ \omega \text{ match M}_2\text{) if M} \equiv \text{M}_1, \text{M}_2\ \}$$
$$\mid \{\ \text{(P match M)} \oplus (\ \omega \text{ match } \emptyset\text{) else}\ \}$$

The empty molecule is no explicitly used in HOCL. It appears in the following reduction:

$$\langle\ (\textbf{one P by M if C}), \text{N}, \text{X}\ \rangle \rightarrow \langle\ \text{X}\ \rangle \text{ if (P match N} = \phi) \wedge \phi\text{C} \wedge (\phi\text{M} \equiv \emptyset)$$

When the molecule X represents the empty molecule, this reaction rule produces an empty solution. In another case, when a reaction produces a non-empty molecule, it will be locally consumed, as we explained before. For example, the reaction $\langle$ (**one** x::Int, ω **by** ω), 3 $\rangle \Rightarrow \langle\ \rangle$ presents the following pattern-matching $\{\text{x} \Rightarrow 3, \omega \Rightarrow \emptyset\}$.

Reaction Naming. For the sake of flexibility, we decided to name the reactions rules for enabling the matching and extraction of specific rules. As a reminder, reaction rules are now considered as molecules, so every entity is a molecule using HOCL. It allows to define rules that can be applied on other rules (by matching the name of such rule), in other words, programs modifying other programs. These rules are known as higher-order rules. The syntax to name a reaction rule is:

$$\textit{name} = \textbf{\textit{replace}}\ P\ \textbf{\textit{by}}\ M\ \textbf{\textit{if}}\ C\ .$$

Note that other atoms can be named using pairs *i.e.*, (name:a), it would not be appropriate to use pairs to tag reactions since they would not be able to react with other molecules anymore. The reaction rules are directly named either into the solution with the syntax:

$$\text{name} = \text{M}$$

or either using the **let** operator that keeps the name in the solution. In the following example, the reaction rule incrementing the integer is named *succ*. After an arbitrary number of increments, the reaction rule *stop* removes *succ* from the solution:

$$\textbf{let}\ \text{succ} = \textbf{replace}\ \text{x}\ \textbf{by}\ \text{x} + 1\ \textbf{in}$$
$$\textbf{let}\ \text{stop} = \textbf{one}\ (\text{succ} = \text{x}), \omega\ \textbf{by}\ \omega\ \textbf{in}$$
$$\langle\ 1, \text{succ}, \text{stop}\ \rangle$$

This example illustrates the higher-order of HOCL since the reaction rule *stop* reacts with the reaction rule *succ*. Similarly, the non-determinism in HOCL is also illustrated since the resulting solution may be any integer.

2.3.4 Chemistry-Inspired Models for Coordination

The chemical metaphor has been used as a source of inspiration for the development of new coordination models [20, 23, 25]. The chemical programming models are good candidates to coordinate applications on distributed platforms [23], since they are implicitly parallel and provide a natural way to express autonomic behavior and distribution. The authors use the higher-order chemical language (HOCL) for describing the coordination among different resources during the execution of a Desktop-Grid-based application. In the same vein, the chemistry-inspired model was proposed to express the coordination of autonomic systems in [20, 25]. Autonomic systems are described using molecules and reaction rules for handling any change in the environment, without external intervention.

In these approaches, the high abstraction's level of chemical rules is the main benefit of these approaches. Chemical rules can be executed in parallel, without any central coordinator and in a distributed way within Grid infrastructures.

Chemically Coordinated Service Interactions

In the following, we introduce some of the most important chemistry-inspired initiatives which provide unconventional service coordination systems.

- *Viroli et al* [134, 135] proposed to use a biochemical metaphor for modelling the coordination in service ecosystems. The authors justify the adoption of concepts from biological and chemical metaphors since the chemical metaphor cannot address the aspects of spatial distribution. Originally, the chemical computational model is based on the idea of a single chemical solution. They decided to design a hybrid metaphor where the biology offers the concept of colonies of simple organisms moving from one colony to the other giving birth to a mechanism for the design of distributed systems.

  The proposed model is composed of molecules or substances floating in a given portion of space (chemical solution) and reacting with other substances to produce new ones. The spatial distribution is represented using the concept of *compartment* meaning a space hosting molecules and chemical reactions. The compartiments can be physically distributed (remote location) and are delimited by a membrane that regulates and filters the transfer of substances from one compartment to the other. Such transfer of substances cross membranes is made possible by the use of chemical reactions. Therefore, this biochemical model enables to express services as substances in a compartment interacting independently with others in a distributed way.

  Finally, the authors highlighted the autonomous and dynamic behavior of this system where services can be dynamically injected in a decentralized way. However, this approach although valid is still relatively distant to our approach. It does not describe how the possible coordination structures generated by the interaction among services are expressed thus remaining very abstract. Moreover, chemical reactions are not considered as substances so this model is not higher-order.

- *Caeiro et al.*[43] built a coordination model inspired by the chemical computational model for the workflow execution named Chemical Workflow Execution (CWE).

CWE is a control-driven language that includes some constructs to describe data-driven flows. Control structures such as *AND-join*, *AND-split*, *OR-join* and *OR-split* are supported, as well as some data structures for the processing of data sets like *one-to-one* and *all-to-all*.

This language presents some similarities with the Event-driven Process Chains modelling language and has been used to describe workflows in commercial tools such as SAP/R.3. This language is composed of five elements:

1. **Functions** represent the tasks participating in a workflow.
2. **Connectors** are in charge of the management and processing of data elements. Moreover, control and data composition operators are implemented through connectors.
3. **Events** define the pre-post conditions for Functions and Connectors.
4. **Data elements** represent the different data used during the execution.
5. **Resources** are the computation units that performs the tasks represented by the functions.

All these elements are molecules floating into a chemical solution or moving across different solutions. Like other chemical approaches, a sub-solution acts as a storage space containing molecules that can react among themselves until it becomes inert.

The authors presented an inherently parallel and dynamic workflow language for the execution of workflows in centralized engines, and highlighted the support of dynamicity where workflow patterns can be adapted at run time. However, this CWE model, although inspired by the chemical metaphor, it has more similarities with event-driven process chain models than with previous chemical computational models.

Closer to the work presented in this thesis, we discuss in the following, several service coordination systems [26, 27, 136] in which higher-order chemical models were used as coordination languages. These approaches shown how the HOCL language offers an abstract and generic way of programming service orchestration. Note that, HOCL supports coordination mechanism such as sequential and parallel execution, mutual exclusion, atomic capture, *rendez-vous* and any pattern expressible with Kahn networks [80]. Kahn Process Networks (KPN) are graph-based models for representing parallel programs that are communicated through unidirectional FIFO channels.

- In [99], the authors justified the use of a chemistry-inspired model, based on the lack support for dynamic adaptation in most of current workflow enactors. To overcome this limitation, chemical reactions arise as an alternative way to model this dynamic behavior, where reactions are autonomously triggered according to actual and local conditions. In such a way, the authors used $\gamma - calculus$ as an abstract and higher-level coordination model, that evolves in time according to the changes without any a *priori* decision. Furthermore, they shown through examples how the $\gamma - calculus$ could be also considered as an appropriate language for describing complex workflows and advanced coordination strategies.

- In [26, 27], the authors suggested the utilization of HOCL as a workflow description language where workflows can be expressed as chemical programs following two principles: *"workflow as a solution of services"* and *"services as sub-solutions"*. This model also supports the construction of some coordination structures such as sequence, parallel execution, mutual exclusion and atomicity, what allows to express workflows as pipelines. Based on that, the authors designed a chemical program representing a very simple workflow through reactions and molecules. This approach introduced an implicit parallel and autonomous coordination language for workflow definition.

- More recently, a continuation of the works [27, 26, 99] have been presented in [136]. By applying the same principles as above, authors proposed a chemical framework based on HOCL for service orchestration that supports most of BPEL primitives. Thus, this HOCL framework enables to construct workflows using some primitives like *invoke, reply, receive, sequence, flow, thrown* and *fault handler* originally provided in BPEL. In this model, all the elements are considered as rules or computational resources, and organized into a three-level hierarchical solution: *service level* representing the workflow definition, *workflow level* defining the tasks execution order and *operation level* formally identifying each task. The dynamicity and implicit parallelism are presented as solutions in this approach to tackle the static nature of BPEL definitions.

As a result, despite its dynamic and autonomous behavior, all these approaches remain somewhat abstract, and also only conceptual regarding the specification of workflows. Authors do not mention how complex workflow structures can be expressed using these approaches. Similarly, HOCL is a data-driven language, what limits the definition of some control workflow structures. So, it is essential to provide both control and data driven behavior to HOCL for supporting the construction of a wide variety of complex workflow structures.

2.4 Conclusion

In this chapter, we presented the benefits that can be obtained by using dynamic and loosely coupled coordination models for the service composition. Thus, we shown how the rule-based models provide an event-driven coordination mechanism that supports the dynamic adaptation, as well as allows to model service compositions with a high level of abstraction. Similarly, we listed out the attractivenesses of the tuplespace-based models, such as the loose coupling for service interactions. Finally, we explored and justified why a chemical metaphor-based model, as a combination of rule-based and tuplespace-based approaches, is a good candidate to coordinate service compositions. The chemistry-inspired model appears as a promising paradigm naturally capturing parallelism, dynamics, while providing a high level of abstraction. However, based on the previous experience, there are many improvements that could be incorporated to this model to solve the current drawbacks when coordinating service compositions.

As a consequence, we decided to design a chemistry-inspired model (inspired by these previous works) that supports the construction of complex workflow structures and decentralizes the coordination of the service compositions. Also, a prototype has been

implemented and experimented to establish the viability of our model, as we will detail in Chapter 3 and Chapter 4.

Chapter 3

Decentralized Chemistry-Inspired Workflow Execution

This chapter [1] initiates the description of our solution. Its main contribution are the decentralization of the workflow execution and the support for the construction of complex workflow structures by using a chemistry-inspired coordination model. According to that, we built a decentralized architecture that uses the higher-order chemical language (HOCL) as executable workflow language. As mentioned at the end of the previous chapter, HOCL is a well-featured chemical language that has successfully used to coordinate service interactions.

In the following section, we describe our decentralized architecture for workflow coordination based on a *chemical* framework, illustrating the adequacy of the chemical paradigm to execute workflows. Next, the adequacy expressiveness of the chemical model and language for modelling complex workflow structures are given in Section 3.2. We presents a comparative review of the literature related to the distributed execution of workflows, as is detailed in Section 3.4. In Section 3.5, we focus on the comparison of our proposed chemistry-inspired workflow language against the more used and mature service coordination models. Finally, we draw a first conclusion about our contribution in Section 3.6.

3.1 Architecture

As illustrated by Figure 3.1, the proposed architecture is composed by two core elements, namely **Chemical Web Services (ChWS)** and the **multiset**. A ChWS is a chemical encapsulation of a Web service. It is co-responsible with other ChWSes of the coordination of the execution of workflows. Physically, ChWSes are hosted by some nodes and are logically identified by symbolic names into this multiset. Each ChWS is equipped with three elements, namely:

1. The *service caller* represents the encapsulation of a Web service invocation. The invocation to the effective possibly distant Web service, is encapsulated in a chem-

[1] The work presented in this chapter has been published in the International Conferences [FPT10,FTP11,FTPW11] and in the National Conference [F11]. I am the main authors of the work presented in this chapter.

ical expression readable by a chemical interpreter. The implementation of the Web service itself is not encapsulated, as shown in Figure 3.1. This element offers a late binding mechanism to discover and bind proper Web services at run time.

2. A local storage space containing part of the multiset, *i.e.*, molecules and reaction rules constituting the data and control dependencies related to the coordination of workflow execution.

3. An HOCL interpreter, acting as the chemical local engine executing the reactions according to molecules and reaction rules stored in the multiset, responsible for applying the defined workflow patterns and transferring data and control information to other ChWSes involved in a workflow.

The multiset acts as a space shared by all ChWSes involved in the workflow. It contains the workflow definition and all information needed by ChWSes for a decentralized execution of a workflow. This information combines molecules representing data and ChWSes, rules representing control dependencies of the workflow, and rules for the coordination of its execution, as illustrated by Figure 3.2. Data and control dependencies of the workflow can be defined beforehand using some workflow executable languages, like the well-known BPEL or SCUFL. For instance, a BPEL specification could be translated into a chemical program, as is detailed in Appendix A.1. Even though HOCL is used to describe and execute workflow specifications, our purpose is to show its potential as executable workflow language. To coordinate the execution of the workflow, we also need some additional chemical rules, which are *generic*, *i.e.*, independent of a specific workflow. Section 3.2.4 focuses on these generic rules.

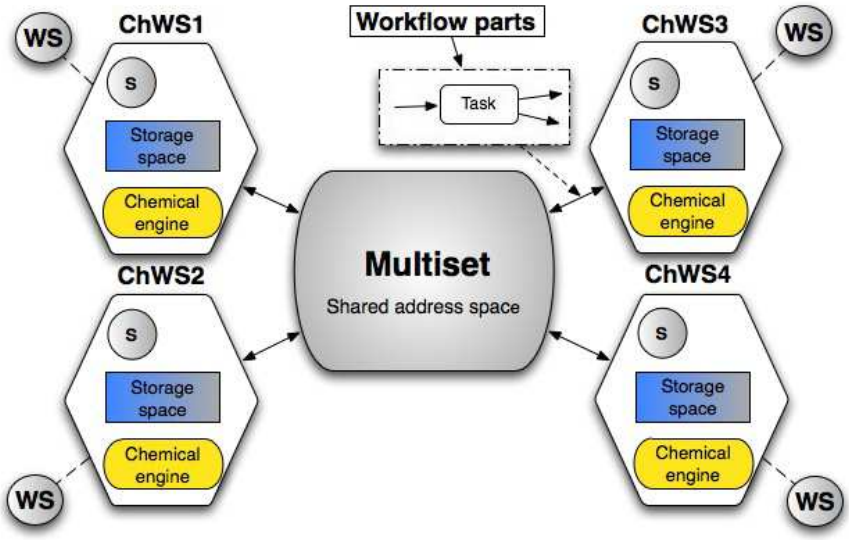

Figure 3.1: The proposed architecture.

The multiset shares some conceptual similarities with the Distributed Shared Memory (DSM) paradigm [109], developed in the area of distributed operating systems. DSM maps a globally unique logical memory address to a local physical memory slot, thus emulating a shared global space on top of a distributed memory platform. By analogy,

multiset mirrors DSM's behavior by exposing molecules and reactions rules physically scattered across a set of ChWSes in a single shared space.

Thus, from a conceptual point of view (illustrated by Figure 3.1), ChWSes communicate through a unique global multiset containing all information needed by ChWSes to execute their part of a workflow. ChWSes exchange data and control dependencies through this multiset. Note that, each ChWS can operate independently of the multiset. In a classical centralized workflow architecture, the services themselves do not know these dependencies, as an engine manages all information and executes coordinates the whole execution.

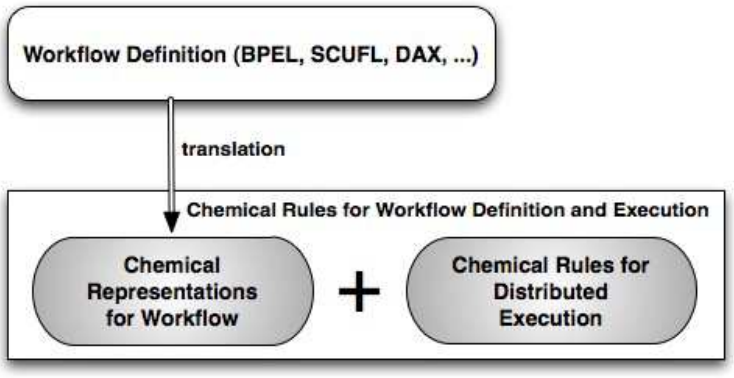

Figure 3.2: Chemical workflow.

From an implementation point of view, the multiset is physically distributed. While apparently, each ChWS only interacts with the multiset, physically, data and control information (molecules and reaction rules of the multiset) are effectively transferred between local multisets (temporary storage spaces) of ChWSes. Put together, the molecules stored by ChWS form the multiset. Figure 3.3 summarizes these two points of view: the upper side shows the conceptual point of view where all ChWSes are *connected* through one multiset; the lower part shows the implementation point of view where all ChWSes are directly interconnected through the multiset, the reactions and molecules being directly transferred from one ChWS to another one using a distributed multiset as detailed in [34]. Figure 3.3 provides a simple example where all ChWS are connected through a sequential workflow (modeled by arrows), but any workflow pattern could be modeled, as we will show in Section 3.2.5. More details of how to implement a decentralized multiset are devised in Chapter 5.

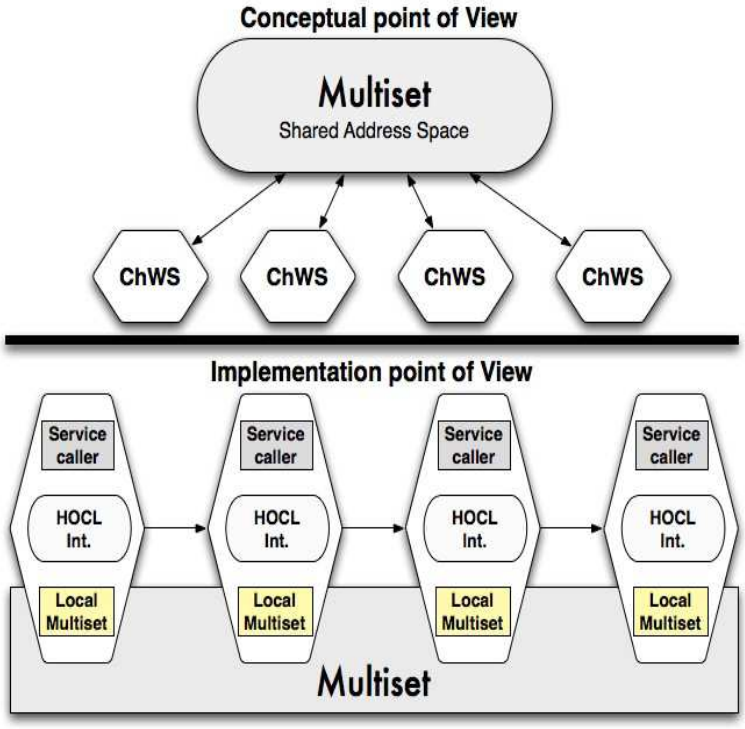

Figure 3.3: Different points of view of the architecture.

3.2 Molecular Composition

Based on the architectural framework presented in Section 3.1, we now focus on the expression of the autonomic and decentralized execution of a wide variety of workflow patterns by defining reaction rules, composing them, and distributing them over the set of services involved in a workflow. Our execution relies on the molecular vision of every entity involved in a workflow execution. In such a way, a new analogy for service composition, in comparison with the existing service composition analogy, was designed, called *Molecular Composition*. In the following, molecules represent the ChWSes themselves, the data they process, their data and control dependencies, and the rules making the whole interact. It is then important to distinct two types of rules inside the multiset: (1) the rules describing data and control flow of a specific workflow to be executed, and (2) workflow-independent rules for the coordination of the execution of any workflow. The latter are referred to as *generic rules* in the following.

In Figure 3.4, an abstract workflow with several services is translated into a molecular composition. Each ChWS disposes of a library of available generic rules. *Some* of them will be used during the execution, depending on the patterns specified in the workflow definition. As we will see in Section 3.3, this composition will react in chain at runtime, performing the execution.

This section is organized as follows. First, Section 3.2.1 explains how a workflow definition will be later executed in a decentralized manner. Section 3.2.2 explains how to define a workflow using the chemical model. Section 3.2.3 determines how global variables

and data processing strategies can be defined using our model. Section 3.2.4 presents the notion of generic rules allowing the decentralized workflow execution. Finally, the combination and distribution of molecules for solving various workflow patterns are given in Section 3.2.5.

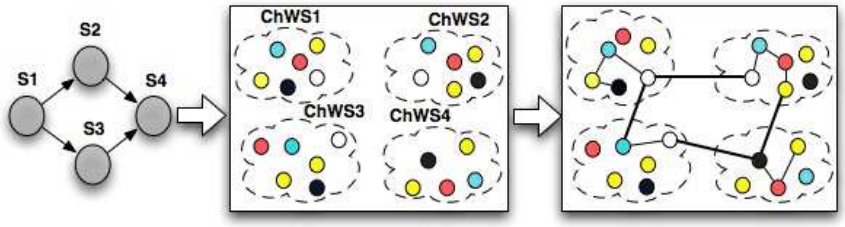

Figure 3.4: Molecular composition from an abstract workflow.

3.2.1 Workflow Partitioning

In our model, a workflow definition is divided into partitions called self-describing workflows, and handled by a light weight workflow management component, called *Chemical Web Services*, located somewhere over the network. Self-describing workflows are partitions of a workflow that carry sufficient control and data information such that they can be processed by a local task execution agent (ChWS) rather than a traditional centralized workflow management system.

Based on this chemical model, the information combines molecules representing data and ChWSes, rules representing data and control dependencies and rules for the coordination of its execution. In our architecture, a workflow specification is translated into a chemical program that is composed of self-describing workflow partitions. Each workflow partition corresponds to each Web service participating in the workflow. Next, each partition will be processed in each ChWS by using the chemical local engines that interpret these self-describing fragments for distributing and executing in a decentralized way. Figure 3.5 summarizes all this process for a workflow, as all the chemical portions are identified and then processed into each ChWS. Once the ChWS has completed, the chemical portion is transferred back to the multiset.

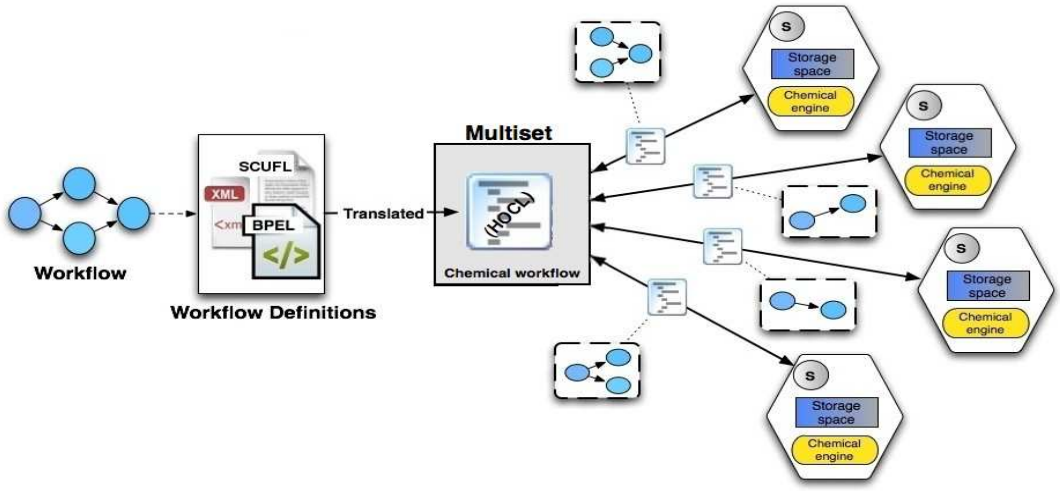

Figure 3.5: Decentralized workflow execution.

Nevertheless, several assumptions have to be taken into account to transform a workflow specification using workflow languages likes BPEL or SCUFL into a chemical program. They are further explained in Appendix A.1.

3.2.2 Chemical Workflow Representation

To express all data and control dependencies of a workflow, we use a series of chemical abstractions inspired by the work in [99]. The general shape of such a representation in Algorithm 1 is as follows: the main solution is composed of as many sub-solutions as we have ChWSes in the workflow. Each sub-solution represents a ChWS with its data and control dependencies with other ChWSes. More formally, a ChWS is one molecule of the form $ChWSi : \langle \ldots \rangle$ where ChWSi refers to the symbolic name given to physical computational device that hosts the ChWSi and hidden its physical location.

Algorithm 1 Chemical workflow representation.

```
1.01  ⟨ // Multiset (Solution)
1.02        ChWSi:⟨...⟩ // ChWS (Sub-solution)
1.03        ChWSi+1:⟨...⟩
1.04        ...
1.05        ChWSn:⟨...⟩
1.06  ⟩
```

Let us consider a simple workflow example illustrated by Figure 3.6. It is composed of four services S_1, S_2, S_3 and S_4. In this workflow, after S_1 completes, S_2 and S_3 can be invoked in parallel. Once S_2 and S_3 have completed, S_4 can be invoked.

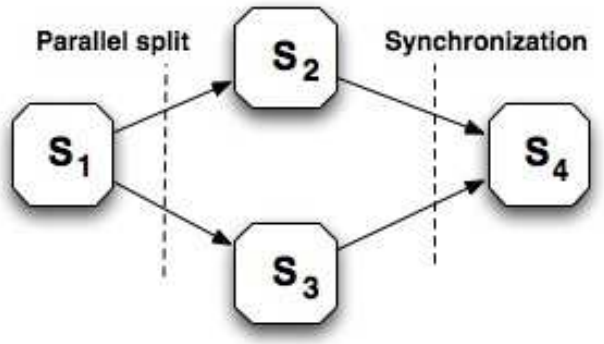

Figure 3.6: Simple workflow example.

The corresponding chemical representation for this workflow is illustrated on Algorithm 2. $ChWS1 : \langle \ldots \rangle$ to $ChWS4 : \langle \ldots \rangle$ represent ChWSes. The links amongst ChWSes are expressed through a molecule of the form DEST:ChWSi with ChWSi being the destination ChWS where some information needs to be transferred. For instance, we can see in the ChWS1 sub-solution that ChWS1 must transfer some information (the result of the invocation of S_1) to ChWS2 and ChWS3 (refer to Line 2.01). Therefore, these links represent the distribution of information whose content is essential to realise the execution.

Algorithm 2 Chemical workflow representation.

```
2.01  ChWS1:⟨DEST:ChWS2,DEST:ChWS3⟩,
2.02  ChWS2:⟨DEST:ChWS4, replace RESULT:ChWS1:value1 by CALL:S2, PARAM:⟨value1⟩ ⟩,
2.03  ChWS3:⟨DEST:ChWS4, replace RESULT:ChWS1:value1 by CALL:S3, PARAM:⟨value1⟩ ⟩,
2.04  ChWS4:⟨replace RESULT:ChWS2:value2, RESULT:ChWS3:value3
2.05          by CALL:S4, PARAM:⟨value2⟩ ⟩
```

Let us focus on the details of the chemical representation of the workflow. As specified by this workflow, ChWS2 presents a data dependency, it requires a molecule RESULT*:ChWS1:value1* containing the result of S_1 to be performed (see the second part of Line 2.02). The two molecules produced by the reaction represent the call to S_2 and their input parameters. They are expressed using a molecule of the form CALL*:Si*, and a molecule PARAM:$\langle in_1,\ldots,in_n \rangle$, where $in_1,\ldots,in_n$ represent the input parameters to call a service *Si*. In Algorithm 2, that input parameter corresponds to the result of some previous service S_1. ChWS3 works similarly.

The chemical language is also able to express control/data-driven flows. Consider ChWS4. It needs to wait until ChWS2 and ChWS3 have completed. This constitutes a control dependency known as *synchronization*. However, as we can see in Line 2.05, the service S_4 is invoked only on *value2* which is the result of S_2. This constitutes a data dependency. The ChWS4 sub-solution contains one reaction rule translating those dependencies in chemical language (see Line 2.05): the presence of molecules RESULT*:ChWS2:value2* and RESULT*:ChWS3:value3* inside the ChWS4 sub-

solution expresses the fulfillment of the control dependencies. The input *value2* inside the PARAM:⟨*value2*⟩ molecule expresses the data dependency in ChWS4. During the execution, as soon as RESULT:*ChWS2:value2* and RESULT:*ChWS3:value3* appear in the ChWS4 sub-solution, the local engine of ChWS4 will be able to perform the reaction that will produce two molecules of the form CALL:*S4* and PARAM:⟨ *value2* ⟩ to call the effective service S_4 on the input *value2*.

To sum up, one reaction rule can express both control and data dependencies. In contrast with the previous *synchronization* pattern, the simple data dependencies are enough to express the simple *parallel split* pattern of S_1 with S_2 and S_3. Thanks to the implicit parallelism of the chemical execution model, the reaction rules inside ChWS2 and ChWS3 can be executed in parallel. Therefore, ChWS2 and ChWS3 will receive the result of S_1 from ChWS1 and the invocation of S_2 and S_3 will take place in parallel.

This fragment of HOCL code is the chemical representation of a workflow, that will be interpreted by chemical local engines, performing the decentralized execution of this workflow thanks to a set of generic rules we introduce in Section 3.2.4 and Section 3.2.5.

3.2.3 Data Manipulation

Global Variables

Global variables in the context of workflows represent pieces of information that need to be read multiple times by the different services involved. In chemical programming, this can be easily implemented through the notion of *multiplets*. Such a molecule can thus be consumed as many times as specified by its multiplicity. A multiplet consists in a specified number of identical molecules. For instance, 3^4 represents 4 instances of the molecule 3. In our context, a molecule m into a main solution of a workflow with a multiplicity n, such that m^n can be consumed in this workflow n times, n being virtually infinite.

Algorithm 3 Variables in a chemical workflow definition.

```
3.01  ⟨
3.02        VAR:"value...", // Global variable
3.03        ...,
3.04        ChWS1:⟨"HOLA"⟩
3.05        ChWS2:⟨...⟩
3.06        ChWS3:⟨...⟩
3.07        ...
3.08  ⟩
```

In Algorithm 3, the tuple of the form VAR:*"value"* defines a global variable that can be accessed by any ChWS in this workflow (Line 3.02). Consider *ChWS1*. The molecule of the form "HOLA" represents a local variable, as it will be consumed in *ChWS1*.

Iteration Strategies

Most of scientific workflow management systems provide a set of *iteration strategies* by defining how input data received from other services are combined together for the computation. They specify how many times the service's task is invoked and what precise combination of input data is given to each of these invocations. Thus, the *dot product* and *cross product* are the most common iteration strategies. We now detail how to support them using our chemical model.

Dot product. A dot product scheme produces tuples of data items with the same position in an arbitrary number of incoming branches of the service. An incoming branch represents the control and/or data dependencies of a service with its predecessors, as shown by Figure 3.7. The service is then launched once for each position, and produces an output located at the same position. The number of items in all input lists or arrays should be the same.

Chemical implementation. For the sake of readability, we here give the rules for a service with two incoming input branches to be composed with a dot product. This can be easily extended to an arbitrary number of incoming branches. A dot product involves a *dotProduct* rule where two molecules representing two lists of atoms are consumed to produce a unique output molecule, DotProduct:⟨ ⟩. Each atom of this molecule corresponds with other atoms located at the same position for each input list. For instance, as detailed in Algorithm 4, the molecule produced by this dot product would be of the form DotProduct:⟨ *("a":1), ("b":2), ("c":3)* ⟩.

Algorithm 4 Dot product example.

```
4.01  let dotProduct = replace List1:⟨ text, ω1 ⟩, List2:⟨ integer, ω2 ⟩, DotProduct:⟨ ω3 ⟩
4.02            by List1:⟨ ω1 ⟩, List2:⟨ ω2 ⟩, DotProduct:⟨ (text:integer), ω3 ⟩,
4.03  in
4.04  ⟨ dotProduct, List1:⟨"a","b","c"⟩, List2:⟨1,2,3⟩, DotProduct:⟨ ⟩ ⟩
```

Cross product. The cross product produces all possible data items combinations from an arbitrary number of incoming branches, each combination being made of one item of each incoming branch. The service task is then launched once for each if these combinations, and produces an output, indexed such that all indices of all inputs are concatenated into a multi-dimensional array.

Chemical implementation. A cross product involves four rules: *crossProduct_start*, *crossProduct*, *crossProduct_list2End* and *crossProduct_end*. Again, we detail the rules for two incoming branches. The *start_crossProduct* rule starts the execution by consuming two molecules representing two lists of atoms, producing two new molecules of the form CrossList1:⟨ ⟩ and CrossList2:⟨ ⟩ which will be used to calculate the cross product and one more molecule of the form CrossProduct:⟨ ⟩, where the temporary cross product result is stored. Then, the *crossProduct* rule iterates over all the atoms of the molecule CrossList2:⟨ ⟩ with the current first atom of the molecule CrossList1:⟨ ⟩. The *list2End* rule is in charge to iterate over all items of the molecule CrossList1:⟨ ⟩.

Finally, the *crossProductEnd* rule determines when all the atoms from the different lists have been consumed, introducing the final result into the solution.

Algorithm 5 Cross product example.

```
5.01  let crossProduct_start = replace-one List1:⟨ text, ω1 ⟩, List2:⟨ integer, ω2 ⟩
5.02                  by List1:⟨ ω1 ⟩, List2:⟨ ω2 ⟩, CrossProduct:⟨ ⟩, CrossList1:⟨ ω1 ⟩,
5.03                  CrossList2:⟨ ω2 ⟩, crossProduct, list2End, crossProductEnd
5.04  let crossProduct = replace CrossList1:⟨ text, ω1 ⟩, CrossList2:⟨ integer, ω2 ⟩, CrossProduct:⟨ ω3 ⟩
5.05                  by CrossList1:⟨text, ω1 ⟩, CrossList2:⟨ ω2 ⟩, CrossProduct:⟨ (text,integer), ω3 ⟩,
5.06  let crossProduct_list2End = replace CrossList1:⟨ text, ω1 ⟩, CrossList2:⟨ ⟩, List2:⟨integer, ω2 ⟩
5.07                  by CrossList1:⟨ ω1 ⟩, CrossList2:⟨ integer, ω2 ⟩, List2:⟨integer, ω2 ⟩
5.08  let crossProduct_end = replace CrossList1:⟨ ⟩, CrossList2:⟨ ⟩, CrossProduct:⟨ ω3 ⟩
5.09                  by ω3
5.10  in
5.11  ⟨ crossProduct, List1:⟨"a","b","c","d"⟩, List2:⟨1,2,3,4⟩ ⟩
```

3.2.4 Chemical Rules for Distributed Execution

As previously mentioned, to ensure the execution of a chemical workflow, additional chemical *generic* rules (*i.e.*, independent of any workflow) must be defined. In addition, for an efficient coordination, these rules use several specific molecules which represents the reactants and products generated during the reactions. Specific molecules allow to manage data related with the transfer of information, condition checking, faults detection and in applying the workflow patterns, in other words, information about the execution. By composing these molecules, complex workflow patterns can be executed in a decentralized way among participants using the chemical paradigm. *How to distribute the workflow patterns responsibilities among participants* is one of the common question that designers of decentralized workflow management systems take in account during the development of their systems. Next, we explain in detail some of these specific molecules, summarized in Table 3.1.

These molecules and rules are included in the chemical local engines and are responsible for the efficient execution of the workflow. We now review three of these *generic* rules, illustrated in Algorithm 6. First, we have rules in charge of the effective invocation of services: *invokes* and *preparePass*. The *invokes* rule invokes a Web service S_i, by consuming the tuples Call:*Si* and Param:$\langle in_1,...,in_n \rangle$ representing the invocation to S_i and their input parameters inside the *ChWSi* sub-solution. The molecule Flag_Invoke indicates whether the invocation can occur. Thus, this execution triggers the call to service S_i (*i.e.*, the service associated with the ChWSi) and produces the result of the service invocation within the solution. In other words, such a rule constitutes an interface between the chemical engine and the service invoked. The *preparePass* rule is used for preparing the transfer of these results to their destination, that will later trigger the execution of the *passInfo* rule.

The rule *passInfo* transfers molecules of information between ChWSes. This rule reacts with a molecule *ChWSj:*⟨Pass:*d:*⟨ω_1 ⟩⟩ that indicates that some molecules (here

Molecules	Definition	Parameters
CALL:Si	Represent the service invocation.	Si: the url where wsdl file is located.
PARAM:⟨ in_1, ...,in_n ⟩	Represent the parameters of a service invocation	in_1,...,in_n: represents all the input parameters for a service invocation.
FLAG_INVOKE	Establish when the service invocation takes place.	–
DISCRIMINATOR	Molecule used to activate a discriminator workflow pattern.	–
MERGE	Molecule used to activate a simple merge workflow pattern.	–
PASS:ChWSi:⟨ ω ⟩	Represent a molecule for the distribution of information.	ChWSi: destination chemical Web service. ω: all molecules to be transferred.
COND_PASS:value	Define the value of one condition.	value: 1 (true) | 0 (false).
COND_PASS:ChWSi:value	Define the value of one condition involving a ChWSi.	ChWSi: ChWS involved in this condition. value: 1 (true) | 0 (false).
ERROR:message	Inform of an error.	message: information about an error.
CANCEL:⟨ ω ⟩	Represent a molecule with the intercepted faults, aborts or error messages.	ω: contains messages about the abort or error in a ChWS.
CANCEL_CHWS:ChWSi	Define the ChWS where the abort/error information should be transferred.	ChWSi: destination chemical Web service.
DEST:ChWSi	Define the destination ChWS for distribution of information.	ChWSi: destination chemical Web service.
SYNCMG_SRC:⟨ ChWSi, ω ⟩	Establish all the ChWS from which a molecule COMPLETED:ChWSi:⟨ ω ⟩ has to be received to start the execution of a destination ChWS. Used in Synchronization merge pattern.	(ChWSi, ω): incomming chemical Web services.
SYNC_SRC:⟨ ChWSi, ω ⟩	Establish all the ChWS from which a molecule COMPLETED:ChWSi:⟨ ω ⟩ has to be received to start the execution of a destination ChWS. Used in Synchronization pattern.	(ChWSi, ω): incomming chemical Web services.
LOCKED:value	Establish when the execution of a reaction rule will be locked even whether it has all the required molecules.	value: 0 (unlocked) | 1 (locked).
RESET:⟨ ω ⟩	Represent a molecule whose content restarts to initial state of one particular solution as many times as it is necessary. Used in Multi merge pattern.	ω: all molecules to store into the solution.
RESULT:ChWSi:⟨ ω ⟩	Contain the outcome of one service invocation for a ChWSi.	ChWSi: chemical Web service already invoked. ω: contains the result of the invocation.
COMPLETED:ChWSi:⟨ ω ⟩	Molecule representing one ChWSi whose execution have been completed.	ChWSi: chemical Web service already invoked. ω: contains the result of the invocation.
SYNCMG_INBOX:⟨ COMPLETED:ChWSi:value, ω ⟩	Represent a molecule which contains all COMPLETED:*ChWSi:value* molecules already consumed. Used in combination with the molecule SYNCMG_SRC:⟨ *ChWSi*, ω ⟩.	ω: represent the rest of molecules COMPLETED:*ChWSi:value* within the solution.
SYNC_INBOX:⟨ COMPLETED:ChWSi:value, ω ⟩	Represent a molecule which contains all COMPLETED:*ChWSi:value* molecules already consumed. Used in combination with the molecule SYNC_SRC:⟨ *ChWSi*, ω ⟩.	ω: represent the rest of molecules COMPLETED:*ChWSi:value* within the solution.

Table 3.1: Specific molecules for the workflow execution.

Algorithm 6 Basic generic rules.

```
6.01  let invokes = replace ChWSi:⟨Call:Si, Param:⟨in1,...,inn ⟩, Flag_Invoke, ω ⟩,
6.02                        by ChWSi:⟨Result:ChWSi:⟨value⟩ , ω ⟩
6.03  let preparePass = replace ChWSi:⟨Result:ChWSi:⟨value⟩ , Dest:ChWSj, ω ⟩
6.04                        by ChWSi:⟨Pass:ChWSj:⟨Completed:ChWSi:⟨value⟩ ⟩ ⟩
6.05  let passInfo = replace ChWSj:⟨Pass:ChWSi:⟨ ω1 ⟩ , ω2 ⟩, ChWSi:⟨ ω3 ⟩
6.06                        by ChWSj:⟨ ω2 ⟩, ChWSi:⟨ ω1, ω3 ⟩
```

denoted ω_1) from ChWSj needs to be transferred to d. These molecules, once inside the sub-solution of d will trigger the next step of the execution. Therefore, the molecule ω_1 will be transferred from sub-solution *ChWSj* to sub-solution *ChWSi*, when reacting with *passInfo* rule.

These rules are the building blocks for decentralized execution. However, they can not, by themselves, solve how to *distribute the workflow patterns responsibilities among participants.*

3.2.5 Solving Workflow Patterns

A set of generic rules for solving complex workflow patterns are now presented, defining the control-logic of the execution. Note that, some of these rules involve every (source and destination) ChWSes participating in a pattern. To understand the following patterns, the meaning of some of the commonly-used terms are graphically explained on Figure 3.7.

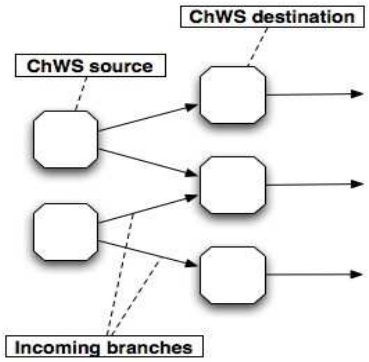

Figure 3.7: Terms.

Parallel split pattern. A parallel split consists of one single thread splitting into multiple parallel threads. (See Figure 3.8).

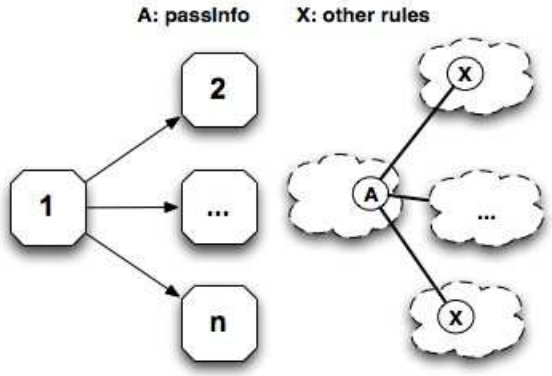

Figure 3.8: Parallel split.

Chemical representation: A parallel split pattern consists in the transfer of molecules produced on one (source) ChWS sub-solution to the others (destination). These reactions will be executed in parallel thanks to the implicit parallelism of the chemical model, so that all the information will be transferred in parallel to ChWSes. The *passInfo* rule has been explained in the Algorithm 6.

Synchronization pattern. A Synchronization pattern is a process where multiple parallel branches converge to one single thread. (See Figure 3.9).

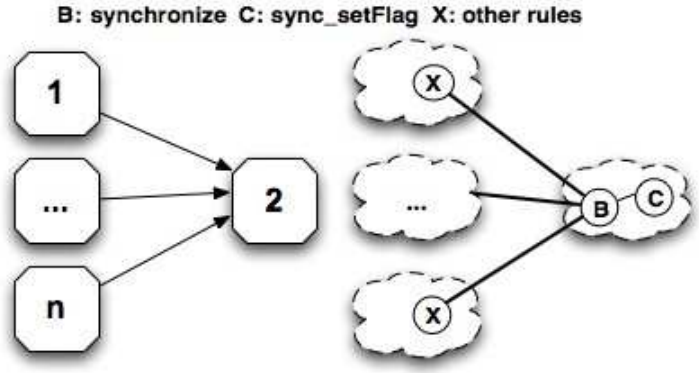

Figure 3.9: Synchronization.

Chemical implementation: A synchronization pattern involves two generic rules described in Algorithm 7. The *synchronize* rule allows to gather all the incoming COMPLETED:*ChWSi:*⟨*value*⟩ molecules, specified by the molecule SYNC_SRC:⟨ *ChWSi,* ω_1 ⟩ representing all the ChWSes from which the destination ChWS needs to receive one molecule COMPLETED:*ChWSi:value* within its solution to trigger its own execution. When all molecules are gathered in the destination ChWS, another reaction, specified by the rule *sync_ setFlag*, is triggered to produce the molecule FLAG_INVOKE allowing the service to be actually called through the *invokes* reaction (Line 7.03).

Algorithm 7 Chemical rules - Synchronization.

7.01 **let** *synchronize* = **replace** SYNC_SRC:⟨ChWSi, ω_1 ⟩, COMPLETED:ChWSi:⟨value⟩, SYNC_INBOX:⟨ ω_2 ⟩
7.02 **by** SYNC_INBOX:⟨ COMPLETED:ChWSi:⟨value⟩, ω_2 ⟩, SYNC_SRC:⟨ ω_1 ⟩
7.03 **let** *sync_setFlag* = **replace-one** SYNC_SRC:⟨ ⟩ **by** FLAG_INVOKE

Molecular composition: synchronize (B) and *sync_setFlag* (C) rules are combined in the destination ChWS.

Exclusive choice pattern. An exclusive choice pattern selects one branch of the workflow among several, based on a condition. (See Figure 3.10).

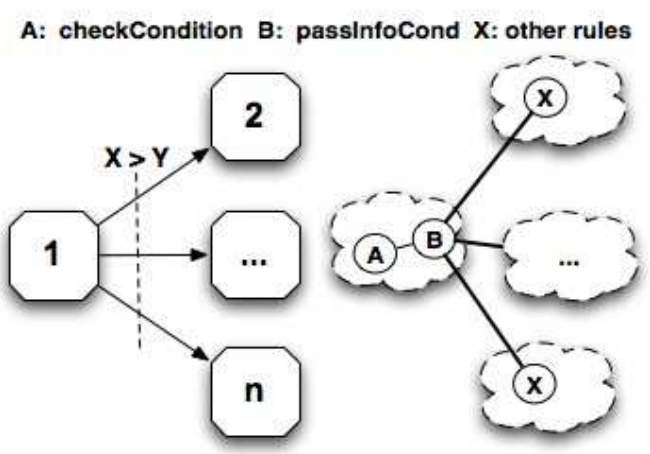

Figure 3.10: Exclusive choice.

Chemical implementation: An exclusive choice pattern involves the *passInfoCond* reaction rule, which is enabled when a given condition has been satisfied. This rule passes the information to the relevant destination ChWS, according to the satisfied condition. The molecule COND_PASS*:1* defines whether the condition has been satisfied. The multi choice pattern, where one or several outgoing branches can be activated depending on a decision process, is also supported by the chemical engines in a similar way, as we will see in the example in Section 3.3.

Algorithm 8 Chemical rule - Exclusive Choice.

8.01 **let** *passInfoCond* = **replace** ChWSj:⟨PASS:ChWSi:⟨ ω_1 ⟩, COND_PASS:1, ω_2 ⟩, ChWSi:⟨ ω_3 ⟩
8.02 **by** ChWSi:⟨ ω_1, ω_3 ⟩, ChWSj:⟨ COND_PASS:1, ω_2 ⟩

Molecular composition: The *passInfoCond* rule (B) will be composed with the dynamic chemical rules in charge of checking the condition (A), transferring the molecule COND_PASS*:1* to the destination ChWSes.

Discriminator pattern. A Discriminator pattern is a structure in the workflow where a service will be activated by the first and only the first completed incoming branch. The subsequent completion of incoming branches will be ignored. (See Figure 3.11).

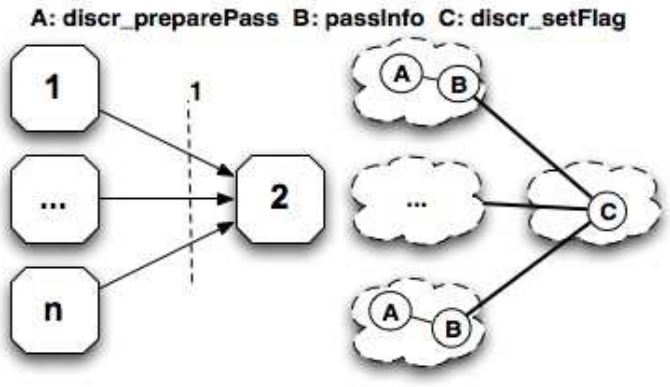

Figure 3.11: Discriminator.

Chemical implementation: As detailed in Algorithm 9, a discriminator pattern involves the *discr_prepare Pass* reaction rule which, on every incoming branch, adds a DISCRIMINATOR molecule to the information to be passed into the destination service (Lines 9.01 and 9.02). The destination ChWS waits for this molecule and only the first DISCRIMINATOR molecule received will react. The FLAG_INVOKE molecule, required to trigger the service invocation, is created (Line 9.03). The following DISCRIMINATOR molecules received will be ignored.

Algorithm 9 Chemical rules - Discriminator.

```
9.01  let discr_prepare Pass = replace DEST:ChWSj, RESULT:ChWSi:⟨value⟩
9.02                           by PASS:ChWSj:⟨COMPLETED:ChWSi:⟨value⟩, DISCRIMINATOR⟩
9.03  let discr_setFlag = replace-one DISCRIMINATOR by FLAG_INVOKE
```

Molecular composition: Each source ChWS has one *discr_prepare Pass* (A) and one *passInfo* (B) rules, they are composed with *discr_setFlag* rule (C) in the destination ChWS.

Simple merge pattern. A simple merge pattern describes the structure where two or more branches converge into a single service with no particular synchronization. The destination service is launched only once. (See Figure 3.12).

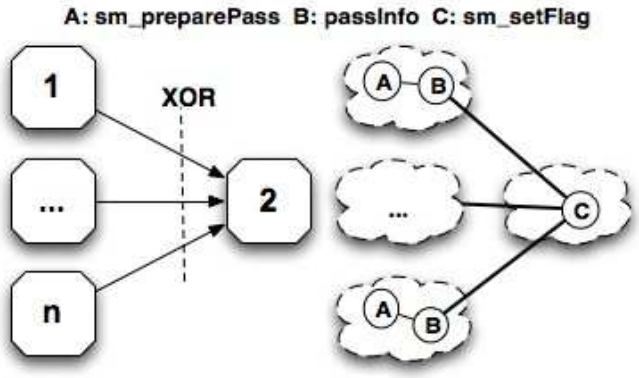

Figure 3.12: Simple merge.

Chemical implementation: A simple merge pattern involves the *sm_ prepareePass* reaction rule which, on every source service, adds a MERGE molecule to the information to be passed into the destination service (Lines 10.01 and 10.03). The destination ChWS waits for this molecule and only the first MERGE molecule received will be consumed. Next, *sm_ setFlag* reaction rule takes place producing one molecule of the form FLAG_INVOKE, allowing the service invocation. Consequently, the subsequent MERGE molecules received will be ignored.

Algorithm 10 Chemical rules - Simple merge.

```
10.01  let sm_ prepareePass = replace DEST:ChWSj, RESULT:ChWSi:⟨value⟩
10.02                         by PASS:ChWSj:⟨RESULT:ChWSi:⟨value⟩, MERGE⟩
10.03  let sm_ setFlag = replace-one MERGE by FLAG_INVOKE
```

Molecular composition: Each source ChWS has one *sm_ prepareePass* (A) and one *passInfo* (B) rules, they are composed with *sm_ setFlag* rule (C) in the destination ChWS.

Synchronization merge pattern. The synchronization merge pattern allows to describe a service for which one or several of its incoming branches can be activated (through a previous multi choice pattern). Then, the synchronization is required when several branches are active. Moreover, a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete. (See Figure 3.13).

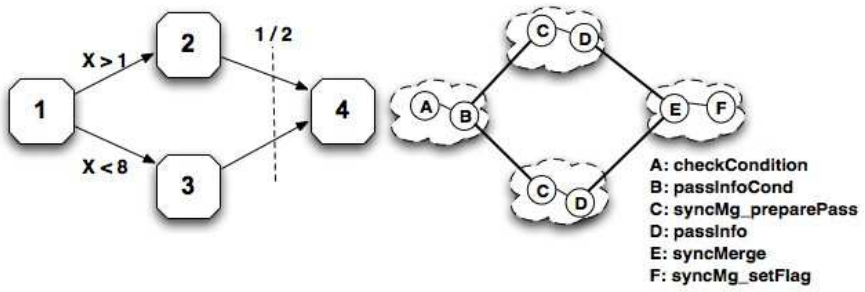

Figure 3.13: Synchronization merge.

Chemical implementation: As detailed in Algorithm 11, a synchronization merge pattern is achieved by transferring one molecule SyncMg_Src:⟨ *ChWSi,* ω ⟩ representing all the ChWSes from which one molecule of the form Completed:*ChWSi:*⟨*value*⟩ has to be received in the destination ChWS. Once, the destination ChWS contains all the needed molecules, it invokes its bounded service. This molecule corresponds to all ChWSes whose branch to the destination ChWS has to be activated, and is generated by ChWS (by the molecule *A* on Figure 3.13). The multi choice pattern is then executed on *service 1*, actually activation of one or both services 2 and 3, through the *passInfoCond* rule. The *syncMerge* rule then waits for the required molecules and finally the *syncMg_ setFlag* rule is triggered, producing a new molecule Flag_Invoke, allowing the invocation. The SyncMg_Inbox:⟨ ω ⟩ molecule stores the already received Completed:*ChWSi:*⟨*value*⟩ molecules.

Algorithm 11 Chemical rules - Synchronization merge.

```
11.01  let syncMg_preparePass = replace Dest:ChWSj, Result:ChWSi:⟨value⟩, SyncMg_Src:⟨ChWSi, ω ⟩
11.02                  by Pass:ChWSj:⟨Completed:ChWSi:⟨value⟩, SyncMg_Src:⟨ChWSi, ω ⟩ ⟩,
11.03  let syncMerge = replace SyncMg_Src:⟨ ChWSi, ω1 ⟩ , Completed:ChWSi:⟨value⟩,
11.04                                  SyncMg_Inbox:⟨ ω2 ⟩
11.05                  by SyncMg_Inbox:⟨Completed:ChWSi:⟨value⟩, ω2 ⟩, SyncMg_Src:⟨ ω1 ⟩
11.06  let syncMg_setFlag = replace-one SyncMg_Src:⟨ ⟩ by Flag_Invoke
```

Molecular composition: The ChWS initiating the multi choice includes a rule to decide on the condition satisfaction, which will be used by the *passInfoCond* (*B*) to activate one or several of its outgoing branches. Then, each intermediate ChWS (encapsulating services 2 and 3) in the example has a *syncMg_preparePass* rule (*C*) and a *passInfo* rule (*D*), composed with *syncMerge* (*E*) and this with *syncMg_ setFlag* rule (*F*) in the destination ChWS (encapsulating service 4, on which the merge should be achieved).

Multi merge pattern. A multi merge pattern is a structure where two or more alternative branches converge again without synchronization into a single subsequent branch such that each enablement of an incoming branch will activate that subsequent branch. In particular, after a multi choice pattern that can lead to several execution scenarios, multi merge will, whatever the number of threads triggered by the multi choice is, merge

the threads into a single one. (See Figure 3.14). Note that this workflow pattern is not supported by BPEL and XPDL-based engines, as devised in [115].

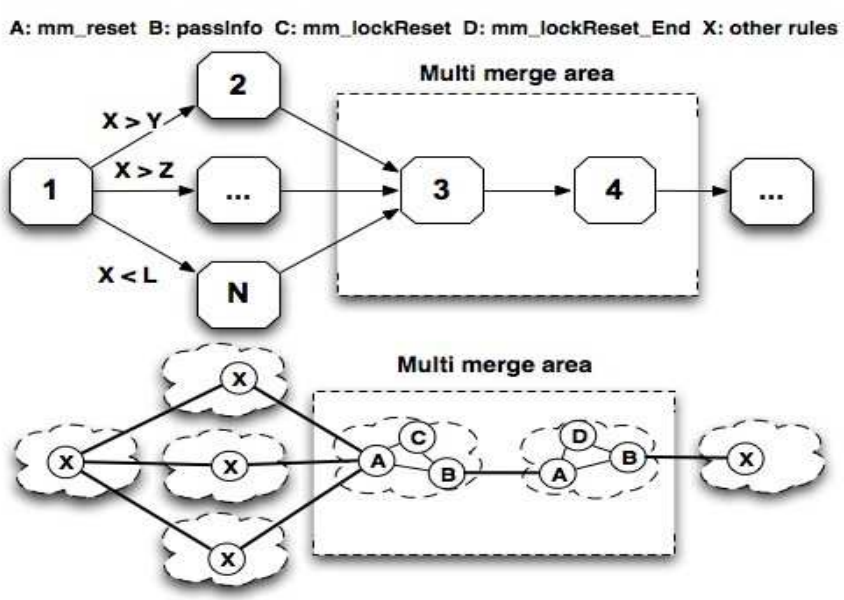

Figure 3.14: Multi merge.

Chemical implementation: As detailed in Algorithm 12, a multi merge pattern involves a set of reaction rules for re-initiating several ChWSes located in a *multi merge area* to their initial state. The *mm_ reset* rule consumes a molecule of the form RESET:$\langle\ \omega\ \rangle$ containing the required information to re-initiate the initial state of one ChWS (all the molecules within its solution before the execution). Once the execution of a given incoming branch has finished and its result has been successfully transferred (through a molecule of the form SUCCESS_PASS:*ChWSi*), the *mm_ lockReset* rule reacts and produces all required molecules to re-initiate the processing of a new incoming request. All ChWSes involved in this pattern include the *mm_ reset* and *mm_ lockReset* rules (Lines 12.01 to 12.04), except the "last" ChWS connecting the *multi merge area* with the rest of the workflow, *i.e.*, *service 4* in Figure 3.14. For the "last" ChWS (*service 4*), the *mm_ reset* and *lockReset_ End* rules are used to re-initiated the initial state or finish the processing of this pattern thus allowing to continue the execution of the rest of the workflow (Lines 12.01 to 12.06).

Algorithm 12 Chemical rules - Multi merge.

```
12.01 let mm_reset = replace-one RESET:⟨ ω ⟩, LOCKED:0, FLAG_INVOKE
12.02                          by ω, RESET:⟨ ω ⟩, LOCKED:1, FLAG_INVOKE
12.03 let mm_lockReset = replace RESULT:ChWSi:⟨value⟩, LOCKED:1, SUCCESS_PASS:ChWSi
12.04                          by mm_reset, LOCKED:0
12.05 let mm_lockReset_End = replace RESULT:ChWSi:⟨value⟩, LOCKED:1
12.06                                by mm_reset, LOCKED:0
```

Molecular composition: All ChWSes participating in this pattern have one *mm_reset* (A) and *passInfo* (B) rules that will be composed with *lockReset_End* rule (D) whether this ChWS represents the exit of our *multi merge area*, otherwise it will be composed with *lockReset* rule (C).

Cancel activity pattern. A cancel activity pattern is the action of withdrawing an enabled task prior to its execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed. To do that, a cancel activity is associated to a specific task at build time (*service 2* in Figure 3.15), thus giving the abitilty to withdraw this task whenever it is required. (See Figure 3.15).

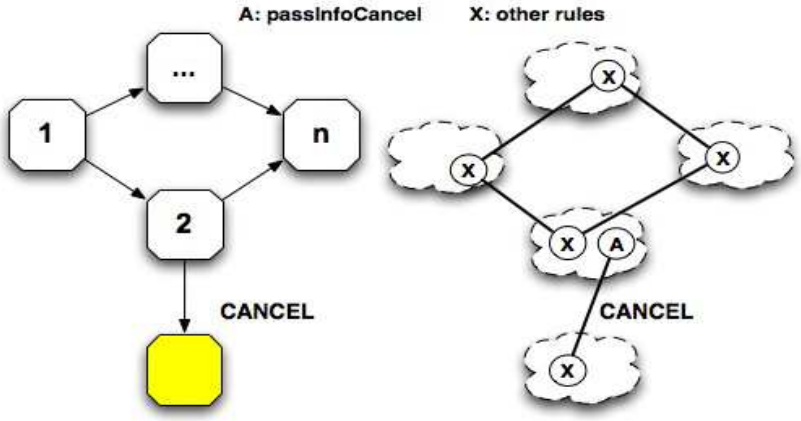

Figure 3.15: Cancel activity.

Chemical implementation: As detailed in Algorithm 13, a cancel activity pattern consists in the transfer of molecules containing error or abort messages produced within the solution of one ChWS, to another ChWS called *CANCEL ChWS*. This *CANCEL ChWS* manages this information and halts the execution. A molecule of the form CANCEL:⟨ ω ⟩ contains the error or abort messages. Similarly, the CANCEL_CHWS:*ChWSj* molecule contains the symbolic name of the *CANCEL ChWS*.

Algorithm 13 Chemical rules - Cancel activity.

```
13.01  let passInfoCancel = replace-one CANCEL:⟨ ω1 ⟩, CANCEL_CHWS:ChWSk, ChWSk:⟨ ω2 ⟩
13.02                       by ChWSk:⟨ ω1, ω2 ⟩
```

Molecular composition: To apply this pattern, one *passInfoCancel* (A) rule is composed with other rules with the aim of handling the withdrawal of a ChWS's execution.

We have shown how the most used workflow patterns can be solved using a set of reaction rules distributed over the services. Virtually all workflow patterns (as for instance those defined in [132]) could be similarly handled.

3.3 Execution Example

For the sake of illustration of the coordination between chemical engines, we present a workflow example, illustrated on Figure 3.16. This figure shows on the top side seven ChWSes applying *parallel split, synchronization, multi choice and discriminator* workflow patterns. On the bottom side, we shows the molecular composition graph representing that workflow. Following the execution, after *ChWS1* completes, it distributes the result to *ChWS2* and *ChWS3* in parallel. Once *ChWS2* and *ChWS3* have been completed, *ChWS4* can react. Next, *ChWS4* checks some conditions and transfers some molecules to *ChWS5* and/or *ChWS6* if they are satisfied. In that way, *ChWS5* and *ChWS6* are connected with *ChWS7* so that some information will be propagated to *ChWS7*. The *ChWS7* will react with the first received molecule from *ChWS5* or *ChWS6*, while the remaining molecules will be ignored. In a composition point of view, we show how each ChWS has a library of molecules where only some of them are composed for executing this workflow. This composition graph omits less important molecules from the composition point of view (data and reaction rules), however the chemical program representing this workflow is available online [2].

[2] http://www.irisa.fr/myriads/members/hfernand/thesisSources/executionExample7services/

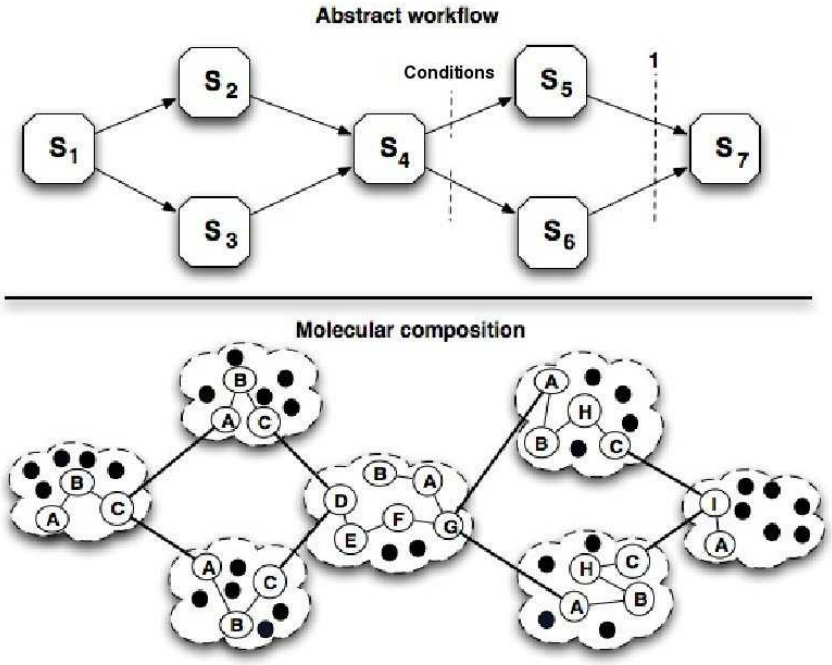

Figure 3.16: Example of coordination.

Note that, thanks to the higher-order property, reaction rules react themselves with other molecules following the composition guidelines. The evolution of the HOCL representation of the workflow is given step by step in Figures 3.17, 3.18 and 3.19. We refer to these three figures all along the section.

Let us first consider the block composed by *ChWS1, ChWS2, ChWS3* and *ChWS4*, as illustrated on Figure 3.17. *ChWS1* has completed through the *invokes* rule (mol. A), producing the result molecule RESULT*:ChWS1:*$\langle value \rangle$. This molecule, through the *preparePass* rule (mol. B), is combined with the molecules DEST*:destination*, preparing the *p*arallel split pattern. Then, the *passInfo* rule (mol. C) triggers it by transferring the outcome of *ChWS1* in parallel.

Once the information is received by *ChWS2* and *ChWS3*, they launch (independently) the *invokes* rule (mol. A), producing two new molecules RESULT*:ChWS2:*$\langle value \rangle$ and RESULT*:ChWS3:*$\langle value \rangle$. The two resulting molecules, through the *preparePass* rule firstly and the *passInfo* rule secondly, will be transferred to *ChWS4* (Lines 15.09 to 15.19), as illustrated on Figure 3.18. Thus, *ChWS4* waits until the completion of *ChWS2* and *ChWS3*, thanks to the rules *synchronize* (mol. D) and *sync_ setFlag* (mol. E).

Consequently, S_4 is invoked producing RESULT*:ChWS4:*$\langle value \rangle$ (Line 15.28). ChWS4 then triggers the reaction rule (mol. F) in charge of checking the conditions of the *multi choice* pattern and transfers some molecules to *ChWS5* and/or *ChWS6* according to the result, thanks to the *passInfoCond* rule (mol. G), as shown by Figure 3.18. For this example, we assume the conditions for *ChWS5* and *ChWS6* are both satisfied in order to apply the *discriminator* pattern.

Let us now focus on the block composed by *ChWS4, ChWS5, ChWS6, ChWS7*, as

illustrated on Figure 3.19 (Lines 16.08 to 16.26). *ChWS5* and *ChWS6* complete and produce their results (Lines 16.08 to 16.21). The *discr_prepare Pass* rules (mol. H) are triggered by the engines of *ChWS5* and *ChWS6*. Two molecules PASS*:ChWS7:*⟨ COMPLETED*:ChWSi:*⟨*value*⟩, DISCRIMINATOR ⟩ are produced (Lines 16.13 to 16.15).

In *ChWS5* and *ChWS6*, the *passInfo* rule (mol. C) propagates the molecule *Pass:ChWS7:*⟨ *information* ⟩ to *ChWS7* (Lines 16.13 to 16.21). As illustrated on Figure 3.18 once they are received by *ChWS7*, the *discr_setFlag* rule (mol. I) is consumed by the first *Discriminator* received, achieving the *discriminator* pattern. Also, it triggers the *invokes* for the invocation of the S_7 producing the final result RESULT*:ChWS7:*⟨*value*⟩ (Line 16.26).

This example completes the description of our solution. We have shown that local engines within ChWSes are co-responsible for applying workflow patterns, invoking services, and propagating the information to other ChWSes. The coordination is achieved as reactions become possible, in an asynchronous and decentralized manner.

3.4 Comparison to Existing Decentralized Approaches

This section is intended to give a more accurate comparison of our approach with some closest and recent works. We observed two methods of distributed coordination approach. In the first one, nodes interact directly. In the second one, they use a shared space for coordination.

Earlier works proposed decentralized architectures where nodes achieve the coordination of a workflow through the exchange of messages [97, 139]. Recently, some works [36, 93, 145] shown the increasing interest in this type of coordination mechanism. In [36], the authors introduce *service invocation triggers*, a lightweight infrastructure that routes messages directly from a producing service to a consuming one, where each service invocation trigger corresponds to the invocation of a service. In [93], an engine is proposed based on a peer-to-peer architecture wherein nodes (similar to local engines) are distributed across multiple computer systems, but appear to the users as a single entity. These nodes collaborate, in order to execute a workflow with every node executing a part of it, as shown by Figure 3.20. Lately, a continuation-passing style, where information on the remainder of the execution is carried in messages, has been proposed [145]. Nodes interpret such messages and thus conduct the execution of services without consulting a centralized engine. However, this coordination mechanism implies a tight coupling of services in terms of spatial and temporal composition. Nodes need to know explicitly which other nodes they will potentially interact with, and when, to be active at the same time. Likewise, a distributed workflow system based on mobile libraries playing the role of engines was presented in [57]. The authors, however, do not give much details about the coordination itself, and about where the data and control dependencies are located.

14.01 ChWS1:⟨Dest:ChWS2, Dest:ChWS3, passInfo, preparePass,**invokes**, * ⟩,
14.02 ChWS2:⟨Dest:ChWS4, passInfo, invokes, preparePass⟩,
14.03 ChWS3:⟨Dest:ChWS4, passInfo, invokes, preparePass⟩,
14.04 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
14.05 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
14.06 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
14.07 ChWS5:⟨Dest:ChWS7, invokes, passInfo, discr_preparePass⟩,
14.08 ChWS6:⟨Dest:ChWS7, invokes , passInfo, discr_preparePass⟩,
14.09 ChWS7:⟨invokes, discr_setFlag⟩

↓

14.10 ChWS1:⟨Dest:ChWS2, Dest:ChWS2, **Result:ChWS1:⟨value⟩**, passInfo, preparePass⟩,
14.11 ChWS2:⟨Dest:ChWS4, passInfo, preparePass, invokes⟩,
14.12 ChWS3:⟨Dest:ChWS4, passInfo, preparePass, invokes⟩,
14.13 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
14.14 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
14.15 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
14.16 ...

↓

14.17 ChWS1:⟨**Pass:ChWS2:⟨Completed:ChWS1:⟨value⟩** ⟩,
14.18 **passInfo**, **Pass:ChWS3:⟨Completed:ChWS1:⟨value⟩** ⟩ ⟩
14.19 ChWS2:⟨Dest:ChWS4, passInfo, preparePass, invokes⟩,
14.20 ChWS3:⟨Dest:ChWS4, passInfo, preparePass, invokes⟩,
14.21 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
14.22 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
14.23 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
14.24 ...

↓

14.25 ChWS1:⟨... ⟩
14.26 ChWS2:⟨Dest:ChWS4, **Completed:ChWS1:⟨value⟩**, passInfo, preparePass, invokes⟩,
14.27 ChWS3:⟨Dest:ChWS4, **Completed:ChWS1:⟨value⟩**, passInfo, preparePass, invokes⟩,
14.28 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
14.29 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
14.30 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
14.31 ...

Figure 3.17: Workflow execution, steps 0-3.

15.01 ChWS1:⟨...⟩,
15.02 ChWS2:⟨Dest:ChWS4, **Result:ChWS2:⟨value⟩**, passInfo, preparePass⟩,
15.03 ChWS3:⟨Dest:ChWS4, **Result:ChWS3:⟨value⟩**, passInfo, preparePass⟩,
15.04 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
15.05 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
15.06 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
15.07 ...

↓

15.08 ChWS1:⟨...⟩,
15.09 ChWS2:⟨**Pass:ChWS4:⟨Completed:ChWS2:⟨value⟩ ⟩, passInfo**⟩,
15.10 ChWS3:⟨**Pass:ChWS4:⟨Completed:ChWS3:⟨value⟩ ⟩, passInfo**⟩,
15.11 ChWS4:⟨Sync_Src:⟨ChWS2,ChWS3⟩, passInfoCond, synchronize, sync_setFlag, invokes,
15.12 preparePass, (replace-one conditions_param by Cond_Pass:1, Dest:ChWS5,Dest:ChWS6
15.13 if (conditions)), Sync_Inbox:⟨ ω ⟩ ⟩,
15.14 ...

↓

15.15 ChWS1:⟨...⟩,
15.16 ChWS2:⟨...⟩,
15.17 ChWS3:⟨...⟩,
15.18 ChWS4:⟨**Sync_Src:⟨ ⟩**, passInfoCond, **synchronize**, **sync_setFlag**, addResult,
15.19 **Sync_Inbox:⟨Completed:ChWS2:⟨value⟩, Completed:ChWS3:⟨value⟩** ⟩,
15.20 Flag_Invoke, (replace-one conditions_param by Cond_Pass:1,
15.21 Dest:ChWS5, Dest:ChWS6 if(conditions)), invokes ⟩,
15.22 ...

↓

15.23 ChWS1:⟨...⟩,
15.24 ChWS2:⟨...⟩,
15.25 ChWS3:⟨...⟩,
15.26 ChWS4:⟨Sync_Src:⟨ ⟩, passInfoCond, synchronize, sync_setFlag, preparePass
15.27 Sync_Inbox:⟨Completed:ChWS2:⟨value⟩, Completed:ChWS3:⟨value⟩ ⟩,
15.28 **Result:ChWS4:⟨value⟩, (replace-one conditions_param by Cond_Pass:1,**
15.29 **Dest:ChWS5, Dest:ChWS6 if (conditions))**⟩,
15.30 ...

Figure 3.18: Workflow execution, steps 4-7

16.01 ChWS1:⟨...⟩,
16.02 ChWS2:⟨...⟩,
16.03 ChWS3:⟨...⟩,
16.04 ChWS4:⟨**Pass:ChWS5:**⟨**Completed:ChWS4:**⟨**value**⟩ ⟩,**passInfoCond**
16.05 **Pass:ChWS6:**⟨**Completed:ChWS4:**⟨**value**⟩ ⟩,**...**⟩,
16.06 ...

↓

16.07 ...
16.08 ChWS4:⟨...⟩,
16.09 ChWS5:⟨**Dest:ChWS7**, **Result:ChWS4:**⟨**value**⟩, invokes, passInfo, **discr_preparePass**⟩,
16.10 ChWS6:⟨**Dest:ChWS7**, **Result:ChWS4:**⟨**value**⟩, invokes, passInfo, **discr_preparePass**⟩,
16.11 ChWS7:⟨invokes, discr_setFlag⟩

↓

16.12 ...
16.13 ChWS4:⟨...⟩,
16.14 ChWS5:⟨**Pass:ChWS7:**⟨**Completed:ChWS5:**⟨**value**⟩,**Discriminator**⟩, **passInfo**⟩,
16.15 ChWS6:⟨Pass:ChWS7:⟨Completed:ChWS6:⟨value⟩, Discriminator⟩, passInfo⟩,
16.16 ChWS7:⟨invokes, discr_setFlag⟩

↓

16.17 ...
16.18 ChWS4:⟨...⟩,
16.19 ChWS5:⟨...⟩,
16.20 ChWS6:⟨Pass:ChWS7:⟨Completed:ChWS6:⟨value⟩,Discriminator⟩,passInfo⟩,
16.21 ChWS7:⟨invokes, **Completed:ChWS5:**⟨**value**⟩, **Discriminator**, **discr_setFlag**⟩

↓

16.22 ...
16.23 ChWS4:⟨...⟩,
16.24 ChWS5:⟨...⟩,
16.25 ChWS6:⟨Pass:ChWS7:⟨Completed:ChWS6:⟨value⟩, Discriminator⟩, passInfo⟩,
16.26 ChWS7:⟨**Result:ChWS7:**⟨**value**⟩, ...⟩

Figure 3.19: Workflow execution, steps 8-12.

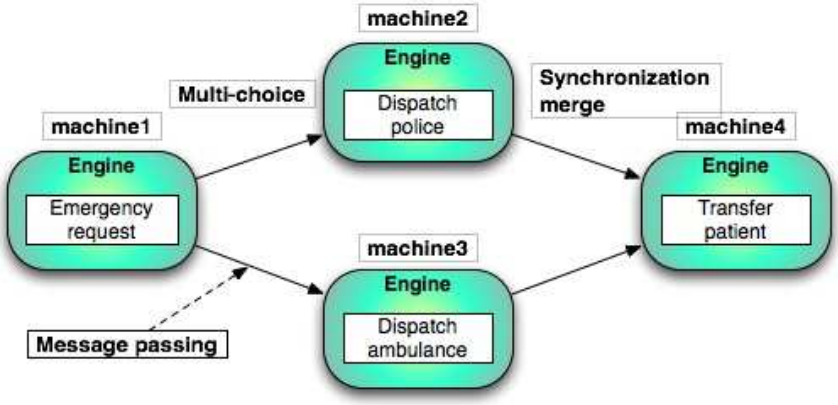

Figure 3.20: Message passing between engines.

Our works deal with the information exchange among ChWSes by writing and reading the multiset which act as a shared space by all ChWSes. Then, the communication can be completely asynchronous since the multiset guarantees the persistence of data and control dependencies. This gives an increased loose coupling to our proposal, and able to deal with dynamic changes in the workflow.

According to this method of distributed coordination, a series of works proposed relying on a shared space a mechanism to exchange information between nodes of a decentralized architecture, more specifically called a *tuplespace* [42, 88, 121]. As detailed in Chapter 2, a tuplespace works as a piece of memory shared by all interacting parties. Thus, using tuplespace for coordination, the execution of a part of a workflow within each node is triggered when tuples, matching the templates registered by the respective nodes, are present in the tuplespace. In the same vein, works such as [100], propose a distributed architecture based on Linda where distributed tuplespaces store data and programs as tuples, allowing mobile computations by transferring programs from one tuple to another. However, the chemical paradigm allows an increased abstraction level while providing support for dynamics.

Using a tuplespace for the execution of workflows, works such as [42],[88] and [121] replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes. In [42] and [88], the authors present a coordination mechanism where the data is managed using a tuplespace and the control is driven by asynchronous messages exchanged between nodes, as shown by Figure 3.21. This message exchange pattern for the control is derived from a Petri net expression of the workflow. In [88], the workflow definition is transformed into a set of activities, that are distributed by passing tokens in the Petri net. However, while in these works, the tuplespace is only used to store data information, our coordination mechanism stores both control and data information in the multiset, which is made possible by the use of the chemical execution model for the coordination of all data and control dependencies.

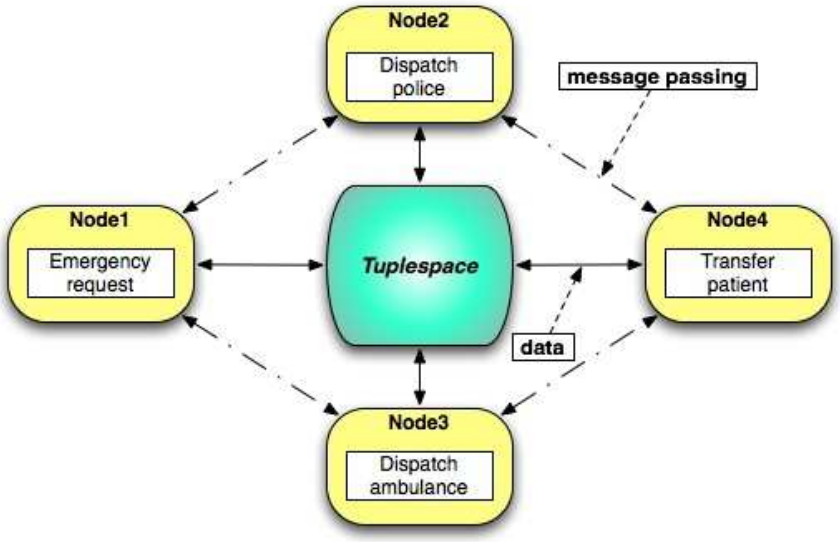

Figure 3.21: Tuplespace data and messages passing control.

Recently, a work [121] uses a shared tuplespace working as a communication infrastructure, the control and data dependencies exchange among processes to make the different nodes interact between them. The authors transform a centralized BPEL definition into a set of coordinated processes using the tuplespace as a communication space. In contrast, the use of BPEL as coordination language hinders from expressing dynamic and self-adaptive behaviors.

3.5 Comparison to Existing Composition Languages

Early workflow executable languages, such as BPEL [101], YAWL [131], or other proprietary languages [96], lack of means to express dynamic behaviors. More recently, approaches were proposed providing this dynamic nature to the service composition. A system governed and guided by rule-based mechanisms supporting dynamic binding and a high level of abstraction for service composition was proposed in [82]. In [46], a BPEL extension based on aspect oriented programming supports the dynamic adaptation of composition at runtime. Other approaches, such as [138], propose coordination models based on data-driven languages like Linda [72], to facilitate dynamic service matching and service composition. However, these approaches rely on architectures where the composition is still managed by a central coordinator node.

Our work share some similarities with the Linda language, as they are both based on a shared space for communication. Nevertheless, the chemical paradigm increases the abstraction level and allows for natural dynamic adaptation.

A more recent series of works address the need for decentralization in workflow execution. The idea they promote is to replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes [42, 121]. These works propose a system based on workflow management components: each workflow component contains

sufficient information such that they can be managed by local nodes rather than one central coordinator. Despite their architectural similarities with our approach, one key difference is that they again use low level abstraction languages such as BPEL or other proprietary languages. In particular, BPEL lacks means to express dynamic behaviors and do not provide concepts for a distributed execution. Similarly, workflow partitioning is a complex task, to be done statically at design time. In other words, there is not any simple algorithm to decentralize the execution of a workflow using BPEL [11, 64]. Some languages have also been proposed for providing a distributed support to service coordination [96, 126]. However, they are finally turned into BPEL for the execution, losing accuracy and expressiveness in the translation. Note that, the scientific workflow languages such as SCUFL, DAX and GWorkflowDL were not considered in this section, as they do not provide support for a decentralized execution.

Our *chemically* inspired approach brings a natural way to express dynamic behaviors and both control and data driven coordinations with a high level of abstraction. Distribution and parallelism being implicit, it finally provide a natural way to construct and execute decentralized workflows.

Recently, works by Viroli *et al.* [135] paved the way for new models of coordination inspired by nature. Our work represents also a concrete step forward in this way, focusing on autonomic workflow execution.

3.6 Conclusion

In this chapter, we presented our first contribution an architecture composed of a shared multiset containing the data and control information needed for coordination, and where several chemical local engines are co-responsible for carrying out the execution of a workflow. The second contribution is a new analogy for service composition, namely *molecular composition*. Such an analogy was shown to be able to express the decentralized and autonomous execution of a wide variety of workflow patterns. Thus, a ready-to-use HOCL library for this purpose has been designed and used successfully.

As a result, a prototype was developed and experimented to validate our approach in the following Chapter 4.

Chapter 4

Implementation and Experimentation

This chapter [1] explores the viability and shows the benefits by using a chemistry-inspired system for service coordination. To do that, we developed three different architectures to implement the concept and ideas mentioned in Chapter 3. These architectures adopt the more representative construction models for the development of workflow management systems. Firstly, we developed a tuplespace-based architecture following the guidelines defined in Section 3.1 of Chapter 3. Secondly, for the sake of comparison and discussion, we also developed two other architectures for a centralized and a fully decentralized workflow execution. Finally, these architectures have been prototyped having in common the HOCL-based workflow engine and the *molecular composition* analogy for the modelling of service interactions.

For the sake of validation, a series of experiments were conducted on our chemistry-inspired workflow system at achieving the following objectives:

- To capture the behavior of our approach when processing different types of workflows.
- To evaluate the benefits of a decentralized coordination compared to using a centralized one when modelling and executing complex workflow structures.
- To establish the viability of a chemistry-based workflow management system in comparison with the more mature workflow management systems (WMS).

The rest of the chapter is organized as follows. Section 4.1 presents the architectural and implementation details of these three architectures. Section 4.2 analyzes the performance results when executing diverse types of workflows. Section 4.3 shows the good properties of the *molecular composition* analogy for expressing and performing some workflow structures in a decentralized manner. Section 4.4 compares the behavior of our system with that from the traditional workflow management systems and two of our prototypes. Finally, Section 4.5 concludes the chapter.

[1] The work presented in this chapter is part of the articles published in the International Conferences [FTP11], the International Workshop [FTPW11] and the Research Report [TRFTP12]. I am the main author of the works presented in this chapter.

4.1 Software Prototype

To validate our approach, we designed and implemented three architectures: a centralized referred to us as *HOCL-C*, a tuplespace-based referred to us as *HOCL-TS* and a fully decentralized referred to us as *HOCL-P2P*. All of them have in common the use of the HOCL-based engine and the *molecular composition* analogy to express workflow structures. The main differences between these architectures are denoted in Figure 4.1 and briefly introduced in the following:

- *HOCL-C.* It is a centralized architecture composed of a unique chemical engine playing the same role of traditional workflow engines.

- *HOCL-TS.* This architecture follows the ideas and concepts proposed in Chapter 3. It is composed of a set of chemical engines collaborating among them through a multiset acting as a tuplespace. Service interactions are loosely coupled as a property inherited by the adoption of the tuplespace model. The computation is now decentralized while the communication remains centralized.

- *HOCL-P2P.* This architecture can be seen as a set of chemical engines directly collaborating among them to execute a workflow. This architecture has many similarities with the work proposed by *Micillo et al.* [93], in which workflow engines are nodes in a P2P network for enabling a fully decentralized workflow management. This architecture is more tightly coupled and needs the distribution of the multiset (workflow definition) prior to the execution.

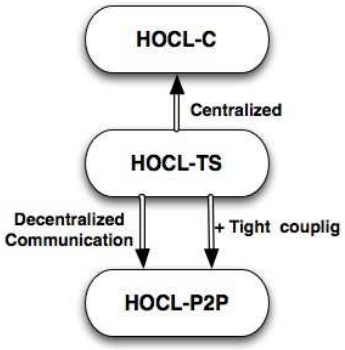

Figure 4.1: Differences between architectures.

Other design and implementation details are further discussed in the following.

4.1.1 Architectures' Design

We now show how the chemical engine can be powered over both centralized and decentralized architectures.

4.1.1.1 HOCL-C

Following the examples of most of workflow management systems mentioned in Chapter 1, the coordination can be managed by a single node, referred to as the *chemical workflow service*, as illustrated by Figure 4.2. First, notice the S components. They represent the interface with the actual distant Web services to be called. Then, the multiset containing the chemical workflow definition and coordination information is accessed by the chemical engine that will perform the reactions required.

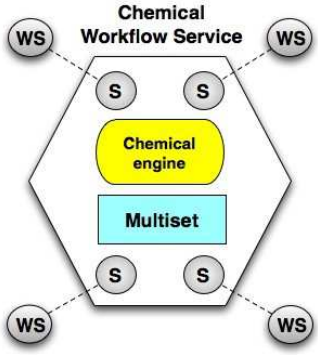

Figure 4.2: HOCL-C WMS architecture.

4.1.1.2 HOCL-TS

Distribute the control means that each service involved will take its part in the coordination process. In this architecture, each Web service is now chemically encapsulated, to form a *Chemical Web Service* (ChWS). There is also as many ChWSes as Web service participating in a service composition. Each ChWS is now equipped with a chemical engine and a *local copy of part of the multiset* on which its chemical interpreter will act, to realize its part of the coordination. The multiset, containing the workflow definition and thus all required coordination information, will now act as a space shared (tuplespace) by all ChWSes involved in the workflow. In other words, ChWSes will communicate by reading and writing it, as illustrated by Figure 4.3. This architecture follows a loosely coupled interaction model, as ChWSes only keep a reference to the tuplespace, instead of having a reference to each ChWS with which they interact. However, the communication remains centralized so that the multiset may become a bottleneck. Recall that in Chapter 5, we will show how to decentralize the multiset itself.

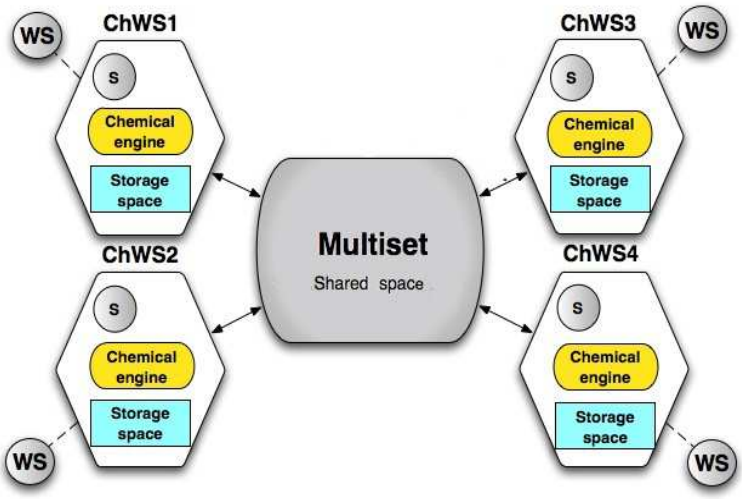

Figure 4.3: HOCL-TS WMS architecture.

4.1.1.3 HOCL-P2P

This framework is similar to the previous architecture, however there is not a multiset working as a tuplespace, computation and communication are fully decentralized.

This architecture is the most common construction regarding the classical decentralized approaches for the workflow execution. Instead of having a unique central workflow engine, a set of engines interact together to execute a service composition in a peer-to-peer system, as proposed in works such as [93, 145]. Accordingly, this architecture is composed of a set of ChWSes relying on message-passing to coordinate the workflow execution, as illustrated by Figure 4.4. This communication mechanism involves the participants in a more tightly coupled interaction, as they have to keep a physical reference to those participants with which they interact.

Like in HOCL-TS, there is as many ChWSes as Web services involved in the workflow. In comparison with HOCL-C, the multiset of each ChWS contains one portion of the workflow definition, instead of having a unique multiset containing the whole specification. These portions will be processed by the chemical engines of each ChWS. Consequently, this architecture assumes that the workflow portions are distributed beforehand and prior to the execution.

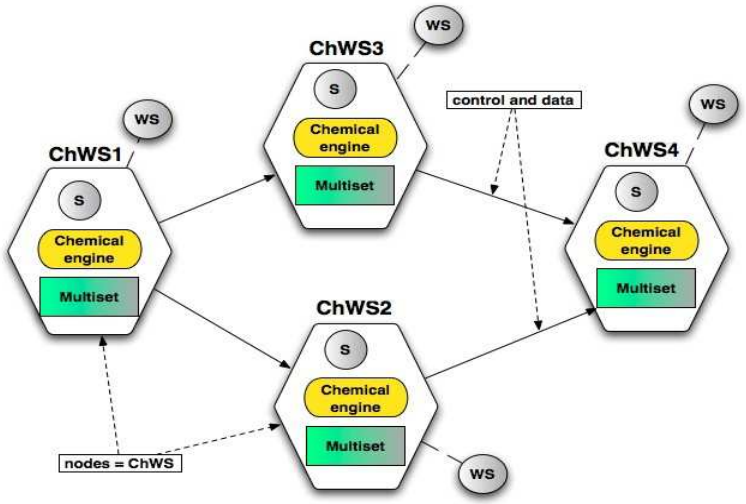

Figure 4.4: HOCL-P2P WMS architecture.

4.1.2 Architectures' Implementation

In this section, we discuss the implementation of three software prototypes for the previously described architectures. The low layer of our prototypes is an HOCL interpreter based on *on-the-fly* compilation of HOCL specifications [110]. The whole prototypes are written in Java.

4.1.2.1 HOCL-C

The prototype is illustrated by Figure 4.5. As mentioned in Section 3.2.2, the workflow definition is executed as a chemical program by the chemical workflow service. The low layer of the architecture is an HOCL interpreter. Given a workflow specification as input (an HOCL program), it executes the workflow coordination by reading and writing the multiset initially fed with the workflow definition. The interface between the chemical engine and the distant services themselves is realized through the *service caller*. The service caller relies on the DAIOS framework [84], which provides an abstraction layer allowing dynamic connection to different flavors of services (SOAP or RESTFul), which abstracting the target service's internals. DAIOS was specially extended with a module which automatically generates dynamic bindings, as well as input and output messages required between the chemical engine and a Web service.

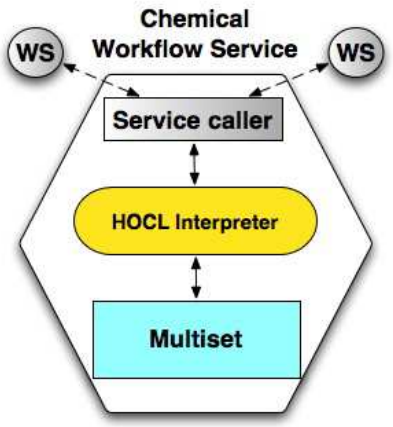

Figure 4.5: HOCL-C implementation.

4.1.2.2 HOCL-TS

The decentralized prototype is illustrated on Figure 4.6. Basically, the difference of this architecture with the centralized implementation is the functionality of the multiset. It now represents a shared space playing the role of a communication mechanism and a storage system. The multiset is initially fed with the HOCL specification of the workflow. More precisely, as we detailed in Section 3.2.2, the workflow definition is comprised of one sub-solution per Web service involved. The information in one sub-solution can only be accessed by the ChWS owner of/represented by that sub-solution.

On each ChWS, a local storage space acts as a temporary container for the sub-solution to be processed by the local HOCL interpreter. The interface between a ChWS and a concrete Web service is realized through the *service caller*, which again relies on the DAIOS framework [84], which provides an abstraction layer allowing to establish dynamic bindings to remote services.

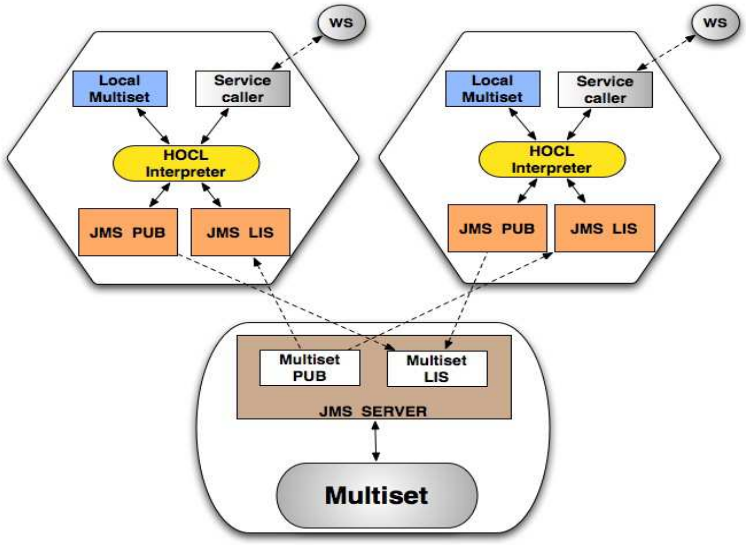

Figure 4.6: HOCL-TS implementation.

ChWSes communicate with the multiset through the Java Message Service (JMS) publisher/subscriber modules. Concretely, we use *ActiveMQ* (version 5.4.1) an implementation of the JMS 1.1 specification, which can be embedded in a Java application server. This ActiveMQ server allows to register and save all the message exchanges between subscribers and publishers. The message exchanged are stored in the server, allowing to be used in the future if a problem arises during the transaction. The multiset is encapsulated into a JMS server to allow concurrent reading and writing operations. The publish/subscribe messaging model is used by the ChWSes and the multiset whereby message producers called publishers pushing each message to each interested party called subscribers. Initially, the *Multiset PUBlisher* pushes the content of each ChWSi solution to each *ChWSes LIStener*. On the ChWS's side, the *ChWS LIStener* receives the content of the ChWSi solution which will be copied into its local multiset. Once the HOCL interpreter is done with its execution, the *ChWS PUBlisher* pushes the content of its sub-solution into the *Multiset LIStener*. These operations models the data flow previously defined in the chemical workflow definition, in particular the DEST molecules specify this data flow among ChWSes.

Recall that this architecture is distributed, a JMS server into the multiset is needed to coordinate all these messages.

4.1.2.3 HOCL-P2P

This decentralized prototype can be seen as a combination of several centralized prototypes interacting among them to execute a workflow, as shown by Figure 4.7. These centralized prototypes are denoted as Chemical Web Services (ChWSes) corresponding each one with each Web service involved in a workflow. As we detailed before, the workflow definition is comprised of one sub-solution per Web service involved. On each ChWS, a multiset contains the information of its specific sub-solution which will be processed by the local HOCL interpreter. Note that, this information is transferred to each ChWS at

build-time. The *service caller* represents the interface between a ChWS and a concrete Web service by relying on DAIOS.

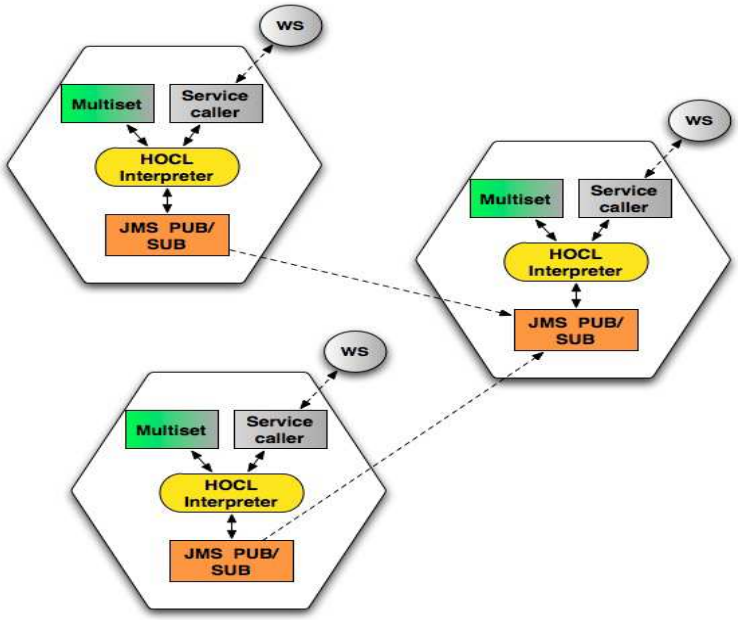

Figure 4.7: HOCL-P2P implementation.

ChWSes communicate among them through Java Message Service (JMS) publisher/-subscriber modules to transfer the control and data information. To do that, a JMS server is included into each ChWS to store the information exchanged. Once the HOCL interpreter of a ChWS is done with its execution, the *ChWS PUBlisher* pushes the outcome into the *ChWS LIStener* of its successor ChWSes (references to its successors have to be maintained). Considering a ChWS, its subscriptions to other ChWSes depends on the data and control dependencies between them.

Note that, this prototype has also been successfully applied to the simulation of agile service networks in the context of Global Software Engineering [TFRZ+12].

4.2 Performance Evaluation of HOCL-TS

Our objective is here to better capture the behavior of a decentralized chemistry-based workflow system. More precisely, we analyse the behavior of our *HOCL-TS* prototype, when processing workflows with different characteristics regarding the number of tasks involved, the amount of data exchanged and the complexity of the coordination required. Experiments were conducted over the nation-wide Grid'5000 platform [8]. More specifically, these experiments were conducted on the *parapide, paramount and parадent* clusters, located in Rennes. The *parapide* cluster is composed of nodes equipped with two quad-core Intel Xeon X5570, 24 GB of RAM; the *paramount* cluster provides nodes with two quad-core Intel Xeon L5148 LV processors, 30 GB of RAM, and the *parадent* cluster is equipped with two quad-core Intel Xeon L5420 processors. All three clusters are furnished with 40GB InfiniBand Ethernet cards.

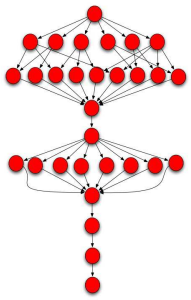

Figure 4.8: 30-task workflow.

4.2.1 Workflows Considered

Three workflows, containing 30, 60 and 100 tasks, were designed inspired by the graph of the Montage workflow [32], a classic astronomical image mosaic workflow processing large images of the sky. Montage workflow combines sequential and parallel flows, making it relevant for such experiments. Our variants of the Montage workflow have different rate of parallelism (number of tasks executed in parallel) and length measured in levels. The *level* of a workflow task is defined as the length of the path leading to it. These workflows are illustrated on Figure 4.8, Figure 4.9 and Figure 4.10, and are respectively referred to as *Workflow30t*, that comprises 30 tasks over 10 *levels*,*Workflow60t*, that comprises 60 tasks dispatched over 13 levels, and *Workflow100t* made of 100 tasks of 19 levels. Our campaign has the following considerations:

1. Each task calls a dummy Web service that basically concatenates strings. This dummy Web service is deployed on a Apache Tomcat [2] server mapped on one machine of the Grid'5000 platform.

2. Tasks at the same level have the same computation cost. Each workflow definition is composed of tasks calling the same dummy Web service.

3. Each task is run by one distinct machine on the Grid'5000 platform.

[2] http://tomcat.apache.org/

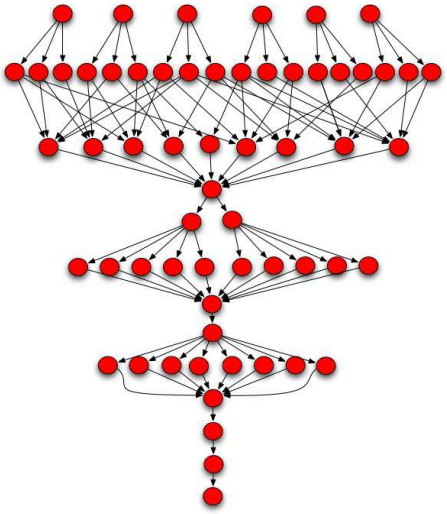

Figure 4.9: 60-task workflow.

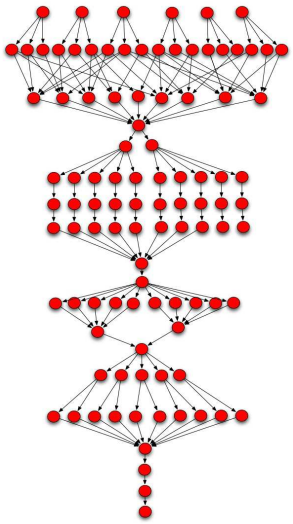

Figure 4.10: 100-task workflow.

Three dummy Web services were built, presenting different rates of data exchanges, for one call of this service, namely 28 bytes for *serviceA*, 583 bytes for *serviceB*, and 3063 bytes for *serviceC*. The definitions used for each workflow are available online [3]. The results of these experiments are averaged over 10 runs.

4.2.2 Managing Large Workflows

Let us first focus on the leftmost bar of the results of each workflow in Figure 4.11, *i.e.*, the completion time of each workflow, but always using *serviceA*.

[3] https://www.irisa.fr/myriads/members/hfernand/thesisSources/workflowHOCL_TS

A first result is that the execution times, for the *Workflow30t* and *Workflow60t*, are quite similar. When looking at the workflows, this can be explained by the fact that when the total number of tasks increases, the parallelism is also increased. There is a higher number of tasks running in parallel, and the number of levels for *Workflow60t* (13 levels) in comparison with the *Workflow30t* (10 levels) is not too high.

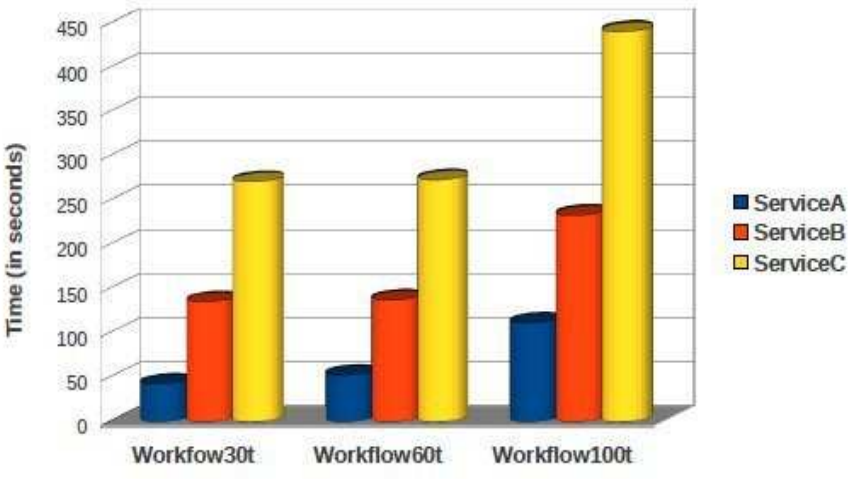

Figure 4.11: Performance results, complexity of workflows and services.

However, the *Workflow100t* shows how a substantial increase of levels (19 levels – more sequentiality) and parallel operations increment the workload of the workflow system responsible to coordinate the execution, as more patterns have to be applied.

To sum up, the length of a workflow inevitably increases the execution time. However, its rate of parallelism does not affect too much to the execution time, as suggested by Figure 4.11. This shows how HOCL-TS adequately deals with high parallelism.

4.2.3 Exchanging Data

For the second experiment, we have dealt with different amount of data exchange. We processed six workflows based on the *Workflow30t* graph, whose tasks are bounded to the same dummy Web service. Note that, for each workflow, we measured the performance using a set of dummy Web services exchanging different amount of data for their execution. This set of services is composed by the three previously-mentioned services and by three other dummy Web services. These services concatenate strings and return objects with different size. Thus, the experiments were conducted with services requiring respectively 28, 583, 3053, 5053, 9773, and 15000 bytes of data exchange. The performance obtained according to the amount of data exchange is illustrated on Figure 4.12.

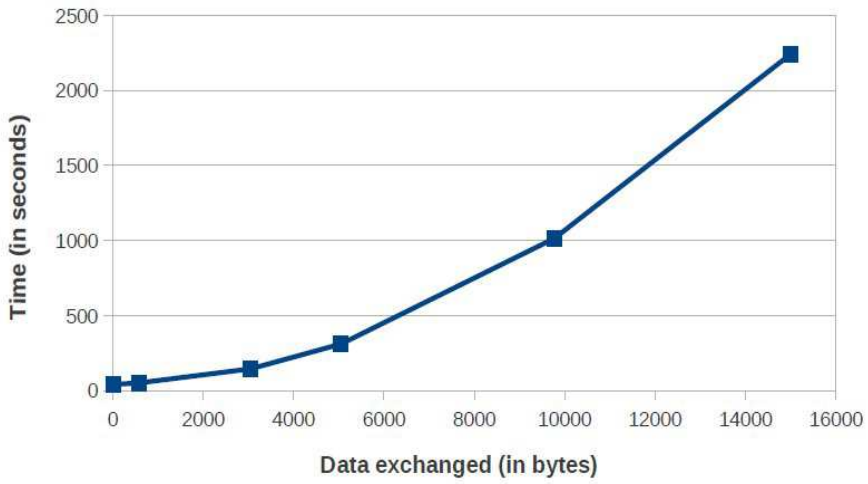

Figure 4.12: Performance results, data exchange.

As we can see in Figure 4.12, that increase in data exchange among tasks provokes an increase of the execution time, suggests a linear degradation of the performance when the size of information exchanged increases. Nevertheless, no bottlenecks have been experienced, even if it may appear with higher data rate. The degradation occurs because the information exchanged is considered itself as a molecule in our chemical model, to be transferred and processed in the multiset. Therefore, further experiments should be conducted to determine how far is the bottleneck.

4.2.4 Workflow's Complexity

This section discusses a different vision of the experiment in Section 4.2.2. We now focus on the evaluation of the workflow *complexity*. Informally, we consider as a complex workflow, a workflow having *many* patterns to be applied and a high rate of data exchange. The complexity also depends on the amount of data exchanged among tasks since these data have to be processed in applying some patterns. The results on the complexity can be deduced by looking at Figure 4.11.

A first observation is that the performance degradation among the three workflows binding their tasks to the different types of services. As we mentioned before, this difference is due to the increment of the amount of data exchange among tasks of any workflow. Thus, the *Workflow30t* using the *serviceC* performs less well than the *Workflow30t* using the *serviceB* or *serviceA*. Secondly, the degree of complexity of the *Workflow100t* in comparison with the *Workflow30t* is high and leads to an important degradation which increases depending on the information transferred among the tasks. However, there is only a slight improvement of the execution time for the *Workflow60t* against the *Workflow30t*, that is explained by the similarity of those workflows in terms of length. Despite the number of tasks participating in the *Workflow60t*, there is more parallelism and only a slight increases of sequentiality, thus reducing the execution time. Finally, *Workflow100t* containing tasks with the *ServiceC* presents an important increase of the execution time, coming from the significant increase of the size and processing time of the multiset.

4.2.5 Discussion

This series of experiments, by offering a proof of concept of the model, while showing its viability in actual deployments, highlights the benefits of a decentralized chemistry-based workflow system. Our workflow engine processes large workflows with a reduced coordination overhead.

However, in our architecture, while the coordination is executed locally on each ChWS (here the coordination is shared among the nodes), the multiset remains a centralized space shared by every ChWSes leading to potential scalability issues. Following this idea, our approach may experience some performance bottleneck when the rate of data exchange becomes very high.

4.3 HOCL-TS vs HOCL-C

To establish a proof of concept on *molecular service composition* explained in Chapter 3, we deployed our *HOCL-C* and *HOCL-TS* prototypes over the nation-wide platform Grid'5000 [8]. This experiment shows the advantages of decentralizing the workflow coordination.

4.3.1 Results

We now present the performance of our approach with both prototypes, using *synchronization*, *parallel split*, *discriminator* and *synchronization merge* patterns, that can be extended in terms of number of services to obtain significant results regarding scalability.

The pattern tested are respectively composed of 5, 15, 30, 45 and 60 tasks, where each task represents a dummy Web service to be executed, each node, in the decentralized prototype, being deployed on a distinct Grid'5000 node. These dummy Web services concatenate strings and return the same data size in contrast with the dummy services used in Section 4.2.2. Results presented are the average results on 6 identical experiments. The *synchronization* pattern consists in one service realizing the synchronization, the others being its incoming branches. The *parallel split* consist in one source node, the others being its outgoing branches. Similar *vertical* extensions have been done for the *discriminator* and *synchronization merge* patterns. In Figure 4.13, n is the number of incoming branches for the *discriminator* pattern and outgoing branches for the *parallel split* pattern.

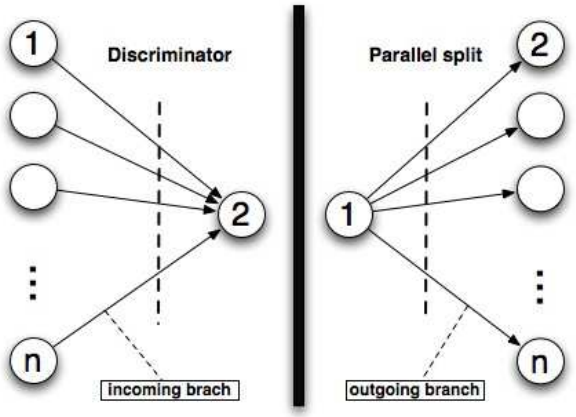

Figure 4.13: Discriminator and Parallel split pattern tests.

In Figure 4.14, performance obtained with the *synchronization* and *parallel split* patterns are given. A first observation is that decentralizing the process brings a non-negligible performance improvement, especially when the number of tasks to coordinate increases, the centralized version suffering from the concentrated workload on the unique coordinator for all the branches.

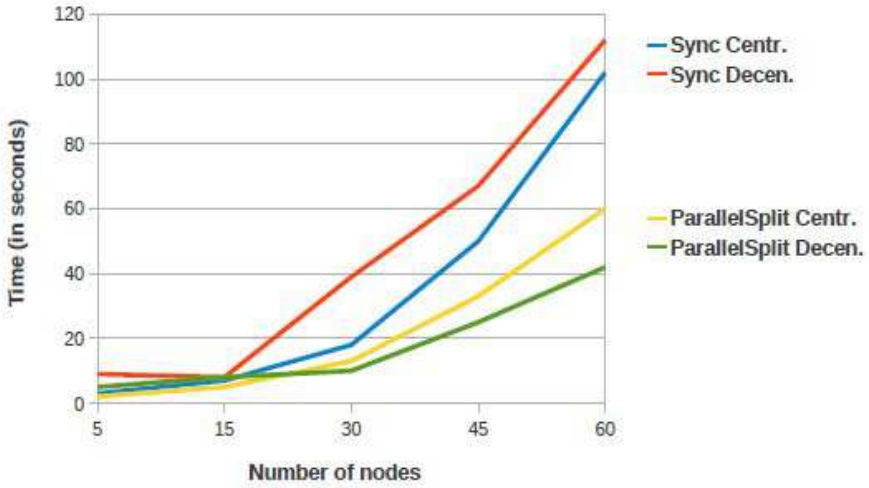

Figure 4.14: Performance on basic patterns.

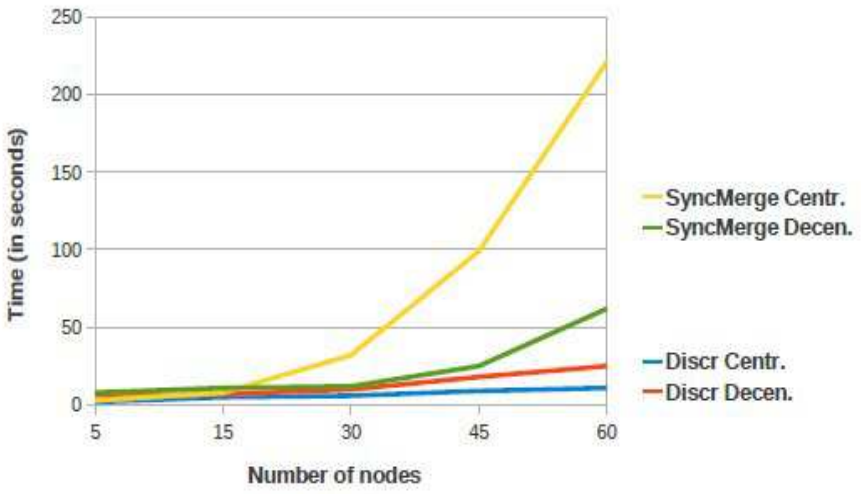

Figure 4.15: Performance on advanced patterns.

Next, we considered the more complex branching and merging concepts used in the *discriminator* and *synchronization merge* patterns. Results are shown in Figure 4.15. A first encouraging result is that the execution time for the *discriminator* shows similar performance evolution for both versions. The *synchronization merge* pattern highlights again the relevance of a decentralized approach, as a significant performance degradation in a centralized environment is again experienced. Again, this can be explained by the complexity of the pattern, composed of a *multi choice* and a *synchronization* pattern, leading to a severe increase in the coordination's workload (on a single node).

4.4 Performance Comparison with Standard WMS

In this section, we present and analyse our experimental results. Five engines have been used: Taverna Workbench 2.2.0, Kepler 2.0. *HOCL-C*, *HOCL-TS* and *HOCL-P2P*.

Recall that our objective is not so much to compare performances, but to establish the viability of a chemistry-based workflow engine. In other terms, Taverna and Kepler represent validated standards we use as guidelines.

4.4.1 Workflows Considered

Three scientific workflows were executed. Illustrated by Figure 4.16 (left), *BlastReport* is a home-built bioinformatics workflow which retrieves a blast report of a protein in a database given its protein ID. The second one, *CardiacAnalysis*, illustrated on Figure 4.16 (right), is a cardiovascular image analysis workflow which extracts the heart's anatomy from a series of image sequences by applying image processing algorithms, developed by the CREATIS-LRMN biomedical laboratory[4]. The third one, *Montage*[5] [32], given in

[4] http://www.creatis.insa-lyon.fr/site/
[5] http://montage.ipac.caltech.edu/

Figure 4.17, is a classic astronomical image mosaic workflow processing large images of the sky.

In order to transform these applications into chemical workflow definitions, we first analyzed their code, exposing their functions or executables as Web services, which will be part of the service composition. Finally, we composed those services based on their control and data dependencies to obtain the final outcome. For instance, the *Cardiac-Analysis* application has an executable script, called *Image_ Pyramid_ Decomposition*, in charge of the creation of three 3D images for a given 3D image. To construct the *Cardiac-Analysis* workflow, this executable was exposed as a Web service named *pyramideDecom* and composed with the other services, as suggested in Figure 4.16 (right).

These workflows present different characteristics related to the number of services involved, the amount of data exchanged and the complexity of the coordination required (data processing included, such as iterations of lists of objects). We attempt to characterize these workflows as follows:

- The *BlastReport* workflow includes 5 services, and presents a medium level of data exchange (simple objects, lists) and low coordination overhead – it is composed mostly of sequences.
- The *CardiacAnalysis* workflow includes 6 services, presenting a high amount of data exchange (complex objects, lists) and a high coordination overhead (synchronizations, loop iteration, parallelism). This overhead does not appear on Figure 4.16 (right). It is due to the re-entrant nature of the services. For each workflow instances, multiple instances of tasks are created from the *interpolation* service to *borderDetection* and *gradient* services (lists of lists of elements to be processed). Indeed, some services produce lists of objects that need to be extracted one by one by iterators, and transferred to the next service asynchronously.
- The *Montage* workflow includes 27 services, and exhibits a low amount of data exchange (simple objects) and medium coordination overhead (parallelism and synchronization patterns).

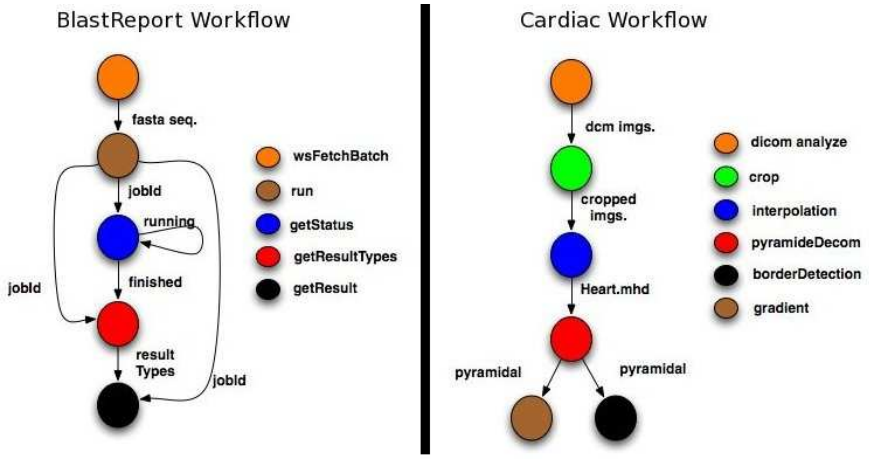

Figure 4.16: BlastReport and Cardiac workflows structures.

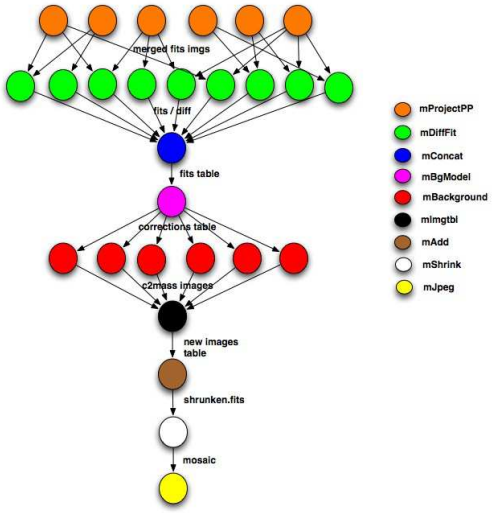

Figure 4.17: Montage workflow structure.

The workflow definitions used for each WMS are available online[6,7] and in Appendix A.2.1, A.2.2 and A.2.3.

4.4.2 Centralized Experiments

The workflows were first run using Taverna, Kepler, and HOCL-C, on a local machine equipped with the Intel core-duo T9600 2.8 Ghz processor and 4GB of memory. Figures 4.18, 4.19 and 4.20 present the results. In Figure 4.18, a first encouraging result is that the execution time for the *Montage* workflow, (*i.e.*, a workflow with limited data exchange and coordination overhead), on Kepler, Taverna and HOCL-C are quite similar, and even slightly reduced on the HOCL-C WMS.

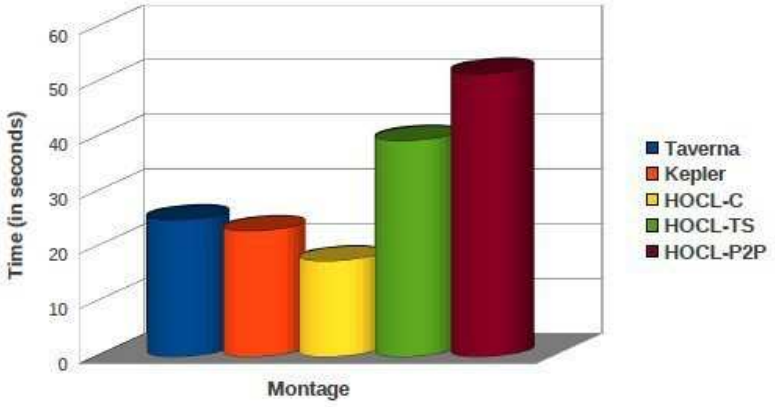

Figure 4.18: Performance results, Montage.

[6] https://www.irisa.fr/myriads/members/hfernand/thesisSources/workflows
[7] http://www.myexperiment.org/workflows/2058.html

For the *BlastReport* workflow on Figure 4.19, while results are again similar for the different WMSes, HOCL-C takes a little more time. This can be explained by the increased size of the multiset for the *BlastReport* workflow (in terms of number of molecules). However, in terms of ratio, execution times remain very close among the *HOCL-** prototypes.

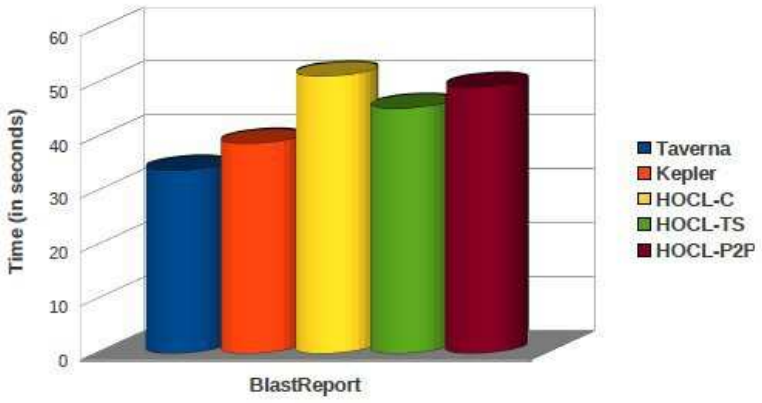

Figure 4.19: Performance results, BlastReport.

Finally, we can see in Figure 4.20 the increased coordination overhead of the *CardiacAnalysis* workflow. As mentioned before, this workflow relies on a lot of data processing related to the coordination itself, which, in the case of HOCL-C, results in a significant increase of the size and processing time of the multiset. Also, no support for parallel execution has been implemented in the HOCL interpreter. These two optimization aspects will be investigated in the future.

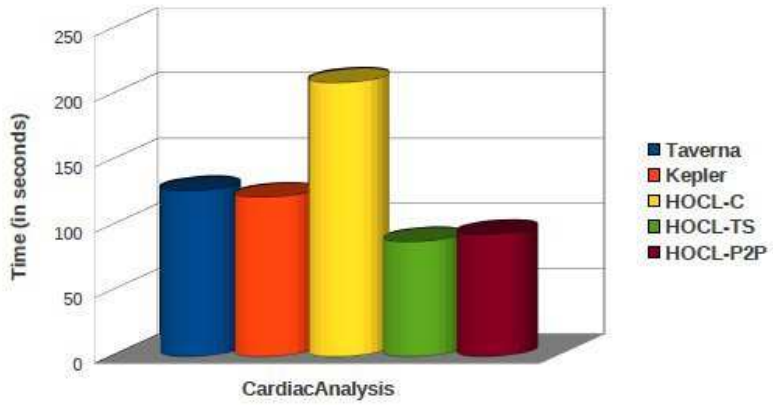

Figure 4.20: Performance results, CardiacAnalysis.

4.4.3 Decentralized Experiments

The workflows were also executed following the HOCL-TS and HOCL-P2P designs. The experiments were conducted on the Grid'5000 platform [8], specifically, on the *adonis*

and *edel* clusters, located in Grenoble, each node being equipped with two quad-core Intel Xeon E5520 processors, 24 GB of RAM and 40GB InfiniBand Ethernet cards. We now focus on the two right-most bars of Figures 4.18, 4.19 and 4.20.

A first observation is that the performance degradation using HOCL-TS and HOCL-P2P on the *Montage* workflow, as illustrated on Figure 4.18. Even though the coordination is executed locally on each ChWS (here the coordination is shared among 27 services in both designs), the time wasted with the network latency to coordinate the nodes is higher than the workload using HOCL-C. On the other hand, HOCL-TS. performs slightly better than HOCL-P2P, what shows how some nodes in HOCL-P2P can lead to some bottlenecks, especially when performing synchronization operations. For instance, when the number of incoming branches increases for a node, its workload can become important. However, further experiments should be conducted.

On the *BlastReport*, a performance gain over HOCL-C is obtained with HOCL-TS and HOCL-P2P, thanks to the distribution of the coordination over the 5 services involved, as shown by Figure 4.19. The *BlastReport* workflow starts to show the benefits by using decentralized prototypes, as an increment of the amount of data exchanged and coordination workload provokes some degradations using centralized architectures. The decentralized prototypes present an acceptable performance in comparison with Kepler and Taverna, as depicted in Figure 4.19. For this workflow, HOCL-TS and HOCL-P2P have similar performance, as there is no synchronization structures.

For the *CardiacAnalysis* workflow, a considerable performance gain is also obtained using HOCL-TS and HOCL-P2P, demonstrating the benefits of a decentralized workflow execution when workflows present a high coordination overhead like *CardiacAnalysis*, which is considered as a computation and data intensive workflow, as depicted in Figure 4.20. Exploiting the processing resources of each ChWS, the list handling and adaptation tasks are separately managed by each ChWS. Therefore, the time wasted with the network latency is now gained by reducing the workload of a central engine. Like *BlastReport*, HOCL TS and HOCL-P2P perform identically due to the lack of synchronization patterns in *CardiacAnalysis*.

4.4.4 Discussion

This series of experiments leads to several conclusions. They constitute a proof of the viability of a chemistry-based workflow engine, as for some representative workflows, its performance are similar and sometimes better to those of Kepler and Taverna. Kepler and Taverna are broadly considered as the defacto standards.

Nevertheless, the network latency comes up as a limitation for decentralized workflow engines when processing workflows such as *Montage*. Its reduced computational load and low rate of data exchange provoke that the coordination time in a decentralized architecture is higher than in a centralized engine, due to the communications (network latency).

These experiments also show how HOCL-TS generally performs slightly better than HOCL-P2P for all the workflows, even if HOCL-TS uses a multiset as communication mechanism. Indeed, another limitation could come from the design of HOCL-TS itself, in which the multiset can constitute a bottleneck. To deal with the decentralization of the multiset itself, we recently formulated solutions based on peer-to-peer protocols, able to distribute and retrieve objects (here, molecules) at large-scale [34]. One of the next

steps of this work is to build the HOCL-TS environment on top of such approaches to remove the bottleneck problem, and proposes a fully decentralized workflow engine.

4.5 Conclusion

Even though there is a wide literature related to decentralized workflow execution, there is no any prototype implementation. Similarly, no proof of concepts for chemistry-based workflow management systems were nowadays given. Therefore, this work gives one more step towards the practicability of this formalism in SOA. We decided to prototype the ideas and concepts defined for our chemistry-inspired workflow management system in Chapter 3. Thus, for the sake of validation, we also prototyped a centralized and decentralized architectures to evaluate the behavior of our approach against them. These prototypes based on the HOCL language show how the chemical engine can be powered over both centralized and decentralized architectures.

A series of experiments were conducted to determine the viability and capture the behavior of our approach when processing workflows with different characteristics. Thus, the plots highlighted the benefits when processing data and computation intensive workflows using decentralized workflow systems, more precisely our approach, in comparison with the more mature and used workflow management systems.

In a more general point of view, this experimental campaign shows the viability of the concept, lifting a barrier on the path to its actual adoption.

Chapter 5

Decentralized Workflow Scheduling through a Chemically Coordinated System

The computation demand of e-Science applications is growing in such proportions that cloud providers are not able to face such required computational power. The high degree of parallelism and the processing of large datasets explain the increasing complexity of these applications, as mentioned in [50]. As a consequence, cloud providers start to cooperate, giving birth to *community clouds*. The cloud providers, in a *community cloud*, federate their resources for allowing the users to execute applications which will be scheduled across multiple cloud sites. For instance, the Venus-C [7] project aims at providing a cloud computing infrastructure for science. The idea of sharing resources, however, carries with it new challenges such as the interoperability between different cloud providers, elasticity, security, as well as economical issues [44, 56].

We have shown in the previous chapter that a chemistry-inspired workflow system is able to decentralize the workflow execution. In this chapter [1], we focus on the crucial feature of scheduling, *i.e.*, the decision making for mapping a job to the more appropriate resource. Accordingly, we here extended our contribution in two ways: (1) providing a chemically coordinated scheduling mechanism for workflows; (2) decentralizing this mechanism via a multiset implemented through a distributed hash table (DHT).

Two primary concerns have to be considered when scheduling applications in such platforms:

1. **Decentralization.** There is a demand for new decentralized coordination mechanism to schedule applications in large-scale environments. Traditional centralized schedulers ought not to be put in use, as they would inevitably suffer from significant reliability and scalability limitations. It is thus essential to promote decentralized and autonomic solutions which embrace all the participants in a community in order to select the appropriate resource for a task. The platform should be enhanced with coordination mechanisms enabling efficient *task-to-resource* mapping.

[1] The work presented in this chapter correspond to the technical report [TRFOT12], and it has been done in collaboration with Marko Obrovac and Cedric Tedeschi.

2. **Cooperation.** A decentralized scheduling system offers a robust solution, however it is not able to handle the cooperation between different resource providers. Cloud providers have to collaborate in order to find the more appropriate resource among the sites in the shortest period of time.

More precisely, we are proposing a fully decentralized workflow scheduling framework including two layers. The top layer is a chemically coordinated shared space where workflows are decomposed into tasks, which are mapped to resources following simple chemical rules. The bottom layer implements this shared space (multiset) in a fully decentralized way, based on a peer-to-peer overlay network allowing the efficient storage and retrieval of molecules. Altogether, the system proposed here, and evaluated through different simulation experiments, is a decentralized framework allowing for an efficient dynamic multiple workflow scheduling.

The rest of the chapter is organized as follow. Section 5.1 introduces the existing scheduling algorithms and systems for workflow scheduling. Section 5.2 describes our decentralized workflow scheduling system and its chemistry-inspired coordination model. Section 5.3 evaluates the performance and network overhead of the framework. Section 5.4 presents the related works, and Section 5.5 draws some conclusions.

5.1 Workflow Scheduling

We generally distinct two types of applications that require to be processed on distributed infrastructures: bag-of-tasks and workflows. In this section, we focus on workflows applications, in which tasks are executed following a data-control flow, *i.e.*, resources have to be available in a certain order. This operation of matching between available resources and application is known as *resource co-allocation*. In the following, we review some of the most used algorithms for resource co-allocation, especially those focused on workflow scheduling, and the most relevant systems for *workflow-resources* matching in distributed infrastructures.

5.1.1 Scheduling Algorithms for Workflows

Today's resource co-allocation systems support the scheduling of workflows, by mapping tasks to resources based on a specific heuristic. A scheduling heuristic is an algorithm that defines which characteristics of a task and resource have to be consider to make an efficient mapping decision.

Generally, two groups of heuristics should be considered when mapping workflows: task-based and workflow-based [37].

5.1.1.1 Task-based Heuristics

This group of heuristics allocates each eligible task to a resource based only about the information of this task (completion time, operative system, etc...). There are a large variety of task-based heuristics, however we now explain those strategies that are being used for workflow scheduling: Opportunistic Load Balancing (OLB) [39], Min-min [77], Max-min [77], and Duplex [39].

- **Opportunistic Load Balancing (OLB).** This heuristic assigns each task to the first available resource without considering any time estimations by running this task on that resource. The main goal of OLB is to keep all resources as busy as possible [39].

- **Min-min.** This heuristic finds for each eligible task (ready to be executed), the resource that gives the *minimum completion* time for this task. Next, the selected task is assigned to that resource for scheduling. This process is repeated with the remaining tasks of the workflow as soon as they are ready for scheduling (when their control-data dependencies have all been satisfied). Note that, the *Min-min* heuristic is more appropriate for workflows whose tasks have a short computation time.

- **Max-min.** This heuristic has some similarities with *Min-min*. Thus, *Max-min* finds for each eligible task, the resource that gives the *minimum completion* time for this task. Next, the task with the overall *maximum completion* time is assigned to the selected resource for scheduling. Finally, this process is then repeated with the remaining tasks as soon as they are ready for scheduling (when their control-data dependencies have all been satisfied). Unlike the *Min-min* heuristic, *Max-min* gives more priority to tasks with longer computation times, so that tasks with a short computation time would wait while longer tasks are executed. Therefore, we noticed that the *Max-min* heuristic may be more appropriate for computation-intense workflows.

- **Duplex.** This heuristic combines the *Min-min* and *Max-min* heuristics. This algorithm performs both heuristics, *Min-min* and *Max-min*, and then selects that one with the best performance. As mentioned in [39], the *Duplex* algorithm suffers a negligible overhead, despite the fact that both heuristics have to be performed.

There are also some other heuristics that belongs to this group such as Minimum Completion Time (MCT), Minimum Execution Time (MET), Tabu, A* and Genetic algorithms. Nevertheless, they are usually not applicable to workflow scheduling strategies since their low efficiency rate [39].

5.1.1.2 Workflow-based Heuristics

This group of heuristics searches for the more appropriate resource selection for the whole workflow. Workflow-based strategies supports variations in computation and communication costs for heterogeneous environments. They assign tasks to resources by considering the workflow structure and the pair processing-communication cost.

In this section, we briefly describe three of most used heuristics in this group: Levelized Minimum Time (LMT) [78], Heterogeneous Earliest-Finish-Time (HEFT)[127] and Dynamic Level Scheduling (DLS) [120].

- **Levelized Minimum Time (LMT).** This heuristic consists in two phases: level sorting and *Min-Time*. First, the LMT algorithm clusters the tasks that have to be executed in parallel, and then order these tasks based on their data and control dependencies, *i.e.*, level by level. Second, each task is matched to the best available

resource, in other words, to the resource that gives the *minimum completion* time for this task. To do that, this algorithm calculates the average execution time (processing and communication cost) of each task across all the available resources. If the number of tasks is greater than the number of available resources, the tasks with the smallest average time are merged until the number of tasks is equal to the number of resources. Then, the tasks are sorted in reverse order (largest average time) based on the average execution time. Finally, this algorithm assigns each task to the best available resource, in other words, the resource which executes it the fastest.

The intuition behind LMT is to assign largest tasks to the fastest resources, and smallest tasks to slower resources.

- **Heterogeneous Earliest-Finish-Time (HEFT).** This is a three phase algorithm. First, a priority value is associated to each task based on the computation and communication costs to reach the exit node from the position (current level) from this task. Second, the resulting list of tasks is sorted by decreasing order of their values. If two tasks have the same priority values, one of them is selected randomly. Third, the algorithm assigns each eligible task to the available resource that gives the *minimum computation* time for scheduling. This heuristic is one of most used among the workflow scheduling systems.

  The objective behind HEFT is to give higher priority to task on the critical path.

- **Dynamic Level Scheduling (DLS).** This heuristic has some similarities with the HEFT algorithm. This is also a three phase heuristic. First, the DLS algorithm assigns a priority value to each task: Considering all the directed paths to the exit nodes from a task t_i, its priority value is defined as the largest sum of computation times along each path. Unlike the HEFT heuristic, the communication cost is not considered in this phase. Second, the resulting list is kept sorted according to the decreasing order of priority values. Third, each eligible task is matched to the available resource with the *minimum communication* costs and *minimum computation* time. This heuristic has two different implementations for its third phase depending of the presence of heterogeneous or homogeneous resources [120].

  DLS is considered as the slowest algorithm in comparison with LMT and HEFT, as devised in [127].

Even though HEFT, LMT and DLS are commonly used regarding the workflow scheduling, other heuristics could be also considered such as Critical-Path-On-a-Processor (CPOP) [127] and Mapping Heuristic (MH) [127].

5.1.1.3 Summary

Despite the wide variety of existing scheduling heuristics, none of them have been shown to deliver an efficient scheduling algorithm for both types of scientific workflows: data and computation intensive [37, 116]. The selection of an appropriate scheduling algorithm depends on several parameters such as the workflow structure, communication costs and the class of tasks in the workflow.

In addition, all these algorithms are based on values estimated by users (task completion times and communication costs), making the schedulers easily prone to *judgment errors*. Therefore, most of the times, workflow scheduling systems delegate the selection of either heuristic to the users.

5.1.2 Workflow Scheduler Systems

Many efforts toward grid workflow management have been made over the last years. This section details some of most used and mature workflow scheduling systems.

DAGMan [71] is a service provided by Condor [2] that allows the execution of workflows over a Grid platform. It contains a meta-scheduler that dynamically processes a DAG structure definition representing tasks of a workflow. DAGMan transfers to the Condor scheduler the tasks as soon as they are ready to be executed. In addition, the transfer of data among tasks is not supported by DAGMan, and the resource selection is done through basic matchmaking.

GridWay [75] is a meta-scheduler standing on top of Globus [3] services that supports DAG workflow definitions using advanced flow structures. In GridWay, the scheduling decisions are made just-in-time considering the requirements for each task and the information provided by the resource broker component. The scheduling strategies can be modified providing new policies. GridWay provides a module for the task execution management. Data transferred is supported between resources.

GridAnt [16] is a client-side workflow system that has some similarities with DAGMan. It is an extension of the Ant [4] tool for controlling the application building processes. The scheduling is done beforehand and fault tolerance is not addressed. This static decision making involves the risk that the resource selection may be made on the basis of information about resources that quickly become outdated.

Askalon [61] is a grid workflow management system whose final goal is to provide an abstract Grid to the application developers. Askalon provides a set of middleware services that support the development and optimization of scientific workflows on the Grid. The workflows are expressed through an XML-based language (AGWL) [62] and then transformed into simple DAG definitions to be scheduled. In Askalon, performance predictions are made considering predicted task execution times and data transfer times among tasks. These predictions are used by different heuristics to generate most appropriate mappings of single or multiple workflows onto the Grid. Askalon has incorporated three scheduling algorithms which can be used interchangeably: HEFT, a genetic algorithm and a "myopic" just-in-time algorithm acting like a resource broker. Performance optimization can be made based on monitoring and dynamic rescheduling mechanisms.

Iceni [89] is a system for workflow definition and enactment on Grids. The Iceni system is in charge of mapping the abstract workflow definition to a concrete workflow selecting the appropriate resources and afterwards monitor the execution on the mapped resources. Once a workflow scheduling process has been computed, the Iceni system tries to reserve the resources at the desired time by negotiating with the resource provider. Iceni provides four resource selection algorithms: random, best of n-random, simulated annealing, and game theory algorithms.

[2] http://research.cs.wisc.edu/condor/
[3] http://www.globus.org/ogsa/
[4] http://ant.apache.org/

Pegasus [54] is the only one workflow system covering the workflow life cycle that enables the execution of large-scale workflows onto Grids. It relies on existing Grid infrastructures such as DAGMan and Globus Toolkit to provide the necessary information for the resource selection. In order to provide a dynamic scheduling, portions of a workflow can be mapped at a time based on data availability. *Task clustering* is also considered where a number of small-granularity tasks are destined for the same resource. Pegasus also incorporates four static algorithms for the resource selection: random, round-robin, Min-min and HEFT. Moreover, an adaptative scheduling can be done thanks to the use of the MAPE functional decomposition, allowing a just-in-time scheduling. Pegasus is a flexible framework that enables the plugging of a variety of components.

MOTEUR [73] is a workflow engine developed by the Modalis Team[5]. The main feature of its execution model enables data, workflow and services parallelism when processing a workflow definition. In addition, this system provides supports for processing large and dynamic datasets. Recently, MOTEUR has been used to run Taverna workflows on real environments such as Grids for E-SciencE (EGEE) [60] and Grid'5000 [8].

Other approaches. Nowadays, there is an increment of workflow management systems such Triana and Kepler that have plugged their systems to execute workflows onto P2P, Grids or Cloud infrastructures. In such a way, Triana supports GridLab GAT [6] (Grid Application Toolkit) for the job scheduling, data management and security issues when executing workflows on Grids. The mapping of tasks to resources are made at runtime with no optimizations or performance predictions. On the other hand, Kepler is another workflow management system that supports execution on Grids, by defining some actors at the composition time in charge of job execution, job monitoring and service discovery. In Kepler, the scheduling is done just-in-time with optimization based on the monitoring information.

All these workflow scheduling systems are fully *centralized*, providing one workflow scheduler that makes decisions for all tasks.

5.2 Decentralized Shared Space for Workflow Scheduling

Highly dynamic large-scale environments by definition, *community clouds* present certain architectural challenges when designing workflow schedulers, as we mentioned before. Consequently, we now present a fully decentralized just-in-time multiple workflow scheduler whose abstract organization is depicted in Figure 5.1. The scheduling process is shared by a set of *chemical engines* running on every resource machine, that constitute the *entry points* for the workflows which will be locally decomposed first into *levels* then into tasks for later execution.

5.2.1 Preliminaries

In this section, we briefly present the required information to understand our proposed architecture for scheduling purposes.

In particular, peer-to-peer systems will help us to decentralize the multiset and the scheduling itself. Thus, P2P overlay networks arose as a solution to solve these problems

[5] http://modalis.i3s.unice.fr
[6] http://www.gridlab.org/

by building decentralized and self-organizing systems. P2P-based systems are naturally scalable and suited for the development of applications in dynamic and large-scale platforms, as pointed it out by *Iaminitchi and Foster* [67].

There is a wide spectrum of different communication frameworks based on P2P overlay network models. However, all these models are classified in two classes of networks: Unstructured and Structured.

- *Unstructured networks* are arbitrarily-shaped networks in which nodes are joining the network in a flat or hierarchical manner, without any requirement on the topology. This network typically uses flooding-based mechanisms to send queries across the overlay within a given radious (number of hops) from the node initiating a search. Hence, all nodes participate in the searching process by propagating the query to their neighbours. Obviously, these flooding techniques increase the traffic load in the network, as well as offer a poor scalability when handling a high rate of queries and sudden increase in the number of nodes. There are several examples of unstructured P2P systems such as Gnutella [124] and BitTorrent [4].

- *Structured networks* are tightly controlled networks that offers a consistent protocol to ensure the uniform distribution of the data among nodes, as well as an efficient retrieval of data. The most well-known approach for structured networks is the distributed hash table (DHT). In the following, we focus on DHT-based systems for allowing the construction of a decentralized and scalable communication mechanism to connect nodes in an overlay network.

5.2.1.1 Distributed Hash Table

A DHT is a P2P infrastructure that supports the scalable storage and retrieval of data items in a dynamic large-scale network.

DHT builds a logical communication network over a physical network. Accordingly, every machine in the physical network represents a node in the logical one. Consequently, the location of a node in the logical network is calculated using a hash function, usually SHA-1, over its ip address. Nodes maintain a routing table containing the identifiers and ip addresses of its neighbours.

In traditional hash tables, data items are stored in memory based on the *(key,value)* pattern. For instance, a key-value pair with the form ('TheBigLebowski.avi','123.26.12.2') means that the node at the address *'123.26.12.2'* contains the movie file *'TheBigLebowski.avi'*. Considering a P2P DHT-based system, pairs are deterministically spread over the nodes of a network in order to achieve a uniform data distribution. This is made possible by using cryptographic functions. These functions provide a mechanism for an optimal data discovery (given a key), and maintain the integrity of the data with a negligible probability of collision.

The data discovery consists of a *lookup* operation in which a query is routed across the nodes to the node whose identifier is the closest to the given key. There is a vast variety of routing and data organization strategies for DHT-based systems. However, the majority of these systems can guarantee that the complexity to reach any data (*i.e.*, the number of hops) is *O(log(n))*, where n is the number of nodes in the overlay.

Even though there are many different implementations of DHT-based systems, we will focus on those implementations forming a ring-shaped overlay like Chord [123] and

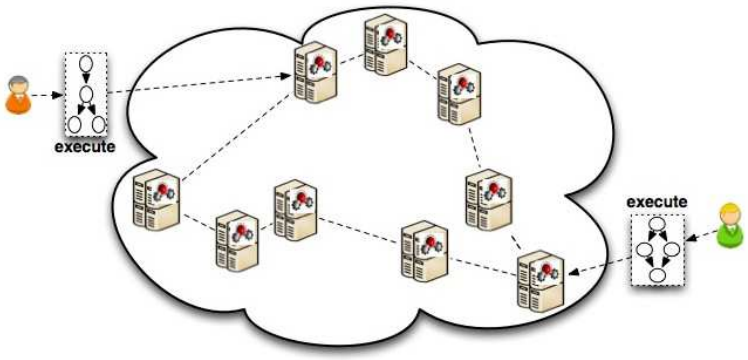

Figure 5.1: Overview of the proposed architecture.

Pastry [114]. Both systems have many similarities, as they mainly differ on the routing process they use. Pastry makes use of a routing strategy, known as Plaxton prefix routing [107]. More precisely, Plaxton prefix consists in choosing as the next hop of a message as the node with an identifier (*nodeId*) that is numerically closest to the key.

However, DHT has one drawback since they only support exact queries. In order to support complex queries, earlier works proposed simple extensions of existing lookup operation. Nevertheless, all these approaches lack efficiency when dealing with *multi-attribute queries* or *range queries*. In such a way, more recently, several approaches [10, 18] have proposed to add and maintain one overlay per type of attribute or dimension. In [18] authors developed a distributed data structure, as a generalization of skip lists referred to as Skip Graphs, that provides the functionality of balanced tree over distributed environments to support *range queries*. On the same vein, P-Grid [10] is another type or data structure that provides a fully decentralized randomized protocol which ensures that the number of hops along the logical path is bounded by *log(n)* of the number of nodes. Furthermore, another approach has been proposed to support *multi-attribute range queries* called Squid [117].

All these approaches increase the complexity of the system depending of the number of overlays per attribute, what consequently brings a number of scalability issues.

5.2.2 Proposed Architecture

As illustrated by Figure 5.2, the proposed system is a two-layer architecture. It takes its roots in a generalized system for decentralized execution of chemical programs [102], but is adapted here for scheduling purposes. We now detail these two layers.

Communication Layer. In order to abstract out the underlying network topology and to deal with the potential unlimited growth of resources, chemical engines are connected in a structured peer-to-peer overlay network [114, 123], illustrated in the lower part of Figure 5.2. Next, we assume that the chemical engines, referred to as *nodes* in the following, communicate through a ring-shaped overlay network. However, indeed, a DHT with a different topology could be used. Connecting the chemical engines into a Pastry *ring* allows the system to cope with both the dynamicity and scalability of the environment. The communication scheme is preserved regardless of whether the number

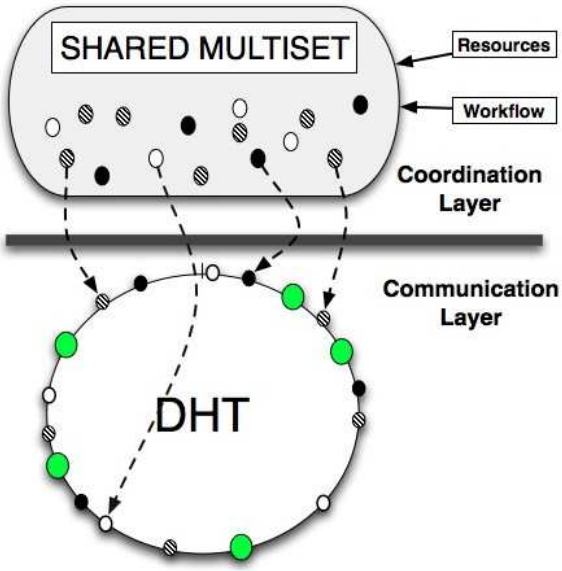

Figure 5.2: Two-layer architecture.

of resources grows or shrinks due to the use of a DHT. Furthermore, because of the symmetric conception of the system, if a resource (or the chemical engine it hosts) fails, another chemical engine can simply resume its work related to scheduling as they all hold the same *generic* scheduling rules.

Coordination Layer. Due to the use of the DHT, chemical engines can share their local data — molecules — with other participants in a scalable fashion (DHTs provide a distribution and retrieval mechanism the complexity of which typically grows logarithmically with the number of nodes). This way, an actual *shared multiset* is created on top of the DHT, to which nodes expose their molecules — levels, tasks and resources, as shown in the upper side of Figure 5.2. Thus, nodes are able to retrieve and consume molecules they do not hold, giving birth to a decentralized *scheduling space*. Moreover, each chemical engine includes workflow-independent rules in charge of workflow decomposition and task scheduling, acting by consuming molecules within the shared multiset. These rules are named *generic scheduling rules* and are explained more in depth in Section 5.2.6.

In the remainder of the section we firstly describe the structure of the molecules of the scheduling space and their placement in the underlying DHT-layer. Then, we detail the decentralized scheduling process carried out as reactions take place in the upper layer.

5.2.3 Scheduling Molecules

There are three types of molecules in our system: molecules representing workflow levels, task molecules and resource molecules. When a molecule is produced, it is assigned a unique identifier based on a cryptographic hash function (SHA1). The molecule is then placed in the shared multiset, *i.e.*, the DHT ring in the bottom layer, by routing it to the appropriate node based on its hash identifier. The molecules and their placement in the DHT are depicted in Figure 5.3.

5.2.3.1 Level Molecules

Upon its entry in the system, a workflow is decomposed into levels by the entry node, producing *level molecules* (white dots in Figure 5.3). A level of a workflow comprises all of the tasks at the same distance from the exit task of a workflow's graph. Level molecules take the form LEVEL : $idLevel$: $\langle task_1, \ldots, task_n \rangle$, where $idLevel$ identifies the level of the workflow, and $task_1, \ldots, task_n$ are the tasks located in it. Each level molecule is then sent to its appropriate destination node according to its hashed value.

5.2.3.2 Task Molecules

Once it is the turn of a level to be processed, the node storing its molecule cuts it into a set of *task molecules* (black dots in Figure 5.3), one per task. A task molecule takes the form TASK : $idTask$: $\langle serv : res\_desc \rangle$: $\langle$DEST : $destTaskId, \ldots \rangle$, where $idTask$ is the task's identifier, $serv$ denotes the actual service to invoke, res_desc is the description of the resource requirements needed to execute the task, and the $\langle$DEST : $destTaskId, \ldots \rangle$ sub-solution specifies to which tasks (given their task identifiers as $destTaskId$) the output of the task has to be sent. Upon a level's decomposition, task molecules are similarly stored in the DHT.

5.2.3.3 Resource Molecules

Physical resources are represented by molecules of the form RES : $idRes$: $\langle feature_1, \ldots, feature_n \rangle$, where $idRes$ is the identifier of the resource and $feature_1, \ldots, feature_n$ are its characteristics, such as the number of processors, the CPU load, or the memory usage. Since the features of a resource variate in time, every so often nodes *republish* resource molecules that replace old ones (the previously published molecule is destroyed and replaced with the new one). The discussion of the modality and the interval of republishing are out of the scope of this work.

These features are used to rank the resources of a community cloud, so that the system can select the appropriate resource for the execution of a task. The ranking criteria are discussed in Section 5.2.6.

Unlike level and task molecules, resource molecules are not hashed using the DHT's cryptographic function. Instead, they are assigned an identifier close to the node which produced them, *i.e.*, where the resource is located, and are, thus, kept on the originating node (as suggested in Figure 5.3). Doing so keeps the network cost of republishing at zero. Moreover, it allows the system to be up-to-date, since when a resource machine crashes the respective molecule consequently disappears from the shared multiset, preventing other participants to try to schedule a task on this resource.

5.2.4 Chemical Nodes

Physically, chemical nodes are deployed on each resource of the community, running as a new process into the machine. Hence, the ring-shaped network overlay itself is composed of as chemical nodes as there are resources. Each node represents a computation unit or coordination object on the overlay, responsible for important operations such the workflow decomposition in levels, the level decomposition in tasks and the scheduling decision making. To do that, some generic rules (*i.e.*, independent of any specific workflow, task

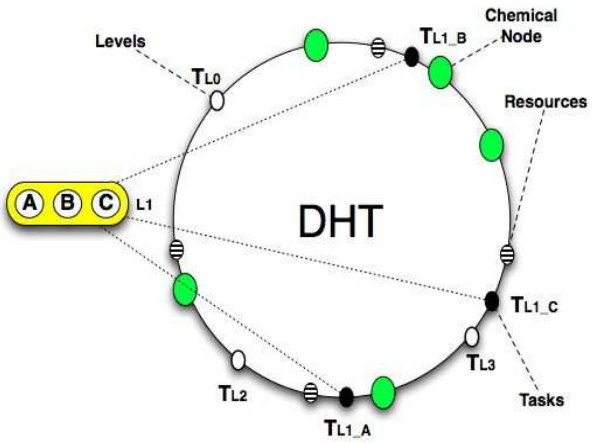

Figure 5.3: Data molecules.

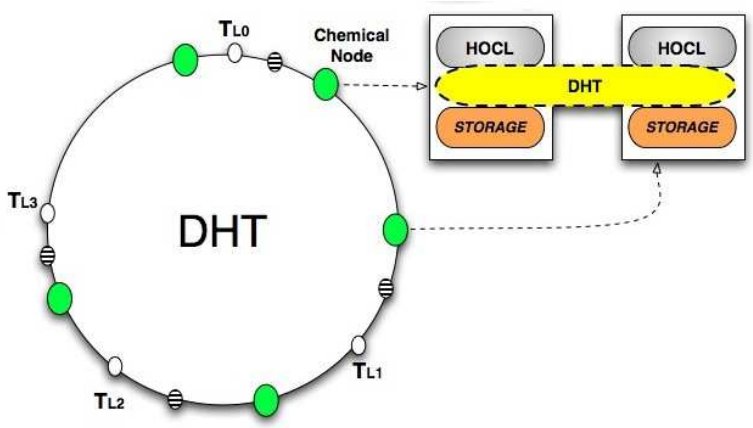

Figure 5.4: Chemical node.

or resource) must be defined. These rules are included in each engine, and are explained in Section 5.2.6.
A chemical node is composed itself of three components (See Figure 5.4) :

- A **Storage space**, containing molecules and rules. The content of the storage space is exposed and shared with the others nodes through the shared multiset.
- An **HOCL interpreter**, working as a chemical engine processing the reactions according to molecules stored in the shared multiset.
- A **DHT overlay**, allowing to be constantly connected to the overlay network accessing to all the information.

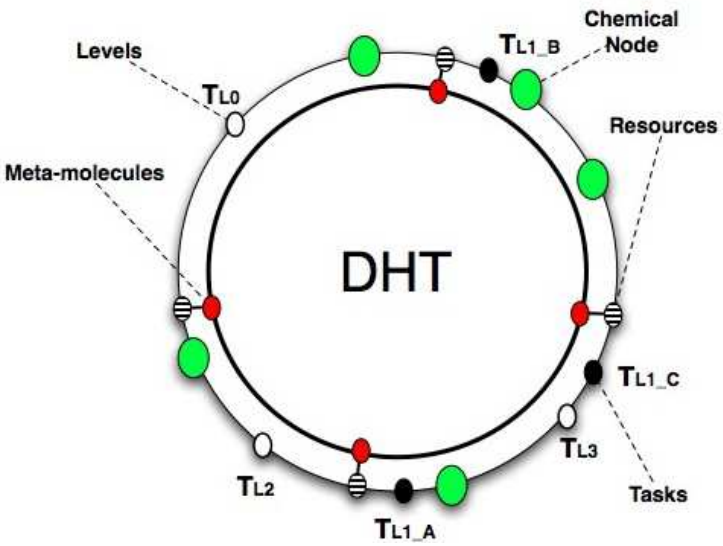

Figure 5.5: Two DHT layer.

5.2.5 Meta-Molecules and Resource Retrieval

For the scheduler to be as efficient as possible, resources have to be ranked in order to match the right tasks with the right resources. Due to the fact that resource molecules stay on their originating nodes, we introduce a second DHT layer, as illustrated on Figure 5.5. It physically matches the original one (*nodeIds* as well as the key space size are preserved), but instead of containing molecules it serves for storing *meta-molecules* — *pointers* to resource molecules, in an order-preserving manner, which means that a meta-molecule's hash identifier is no more cryptographically hashed but based on its molecule's value. Therefore, a meta-molecule *testifies* to the existence of a particular resource molecule; when a node obtains a meta-molecule, it is able to consequently obtain the original resource molecule the meta-molecule derives from.

Meta-molecules are only created for resource molecules. When a node republishes its resource molecule, it creates a matching meta-molecule and stores it in the second DHT layer by routing it to the appropriate node. Meta-molecules are stored in an order-preserving manner, which means that a meta-molecule's hash identifier is based on its molecule's value. As an illustration, consider two resource molecules $M_1 = \text{RES} : cpu(80\%)$ and $M_2 = \text{RES} : cpu(50\%)$. Supposing the ordering criterion for resource molecules is processor utilization, M_1's meta-molecule's hash identifier would be greater than that of M_2's meta-molecule, since the resource represented by M_1 is used more than M_2's resource.

The second, order-preserving DHT layer is used when trying to map a task to a resource. When a node is searching for an appropriate resource to execute its task on, it simply consults the second DHT layer for resource meta-molecules. Moreover, the node is able to precisely locate specific matching resources in this layer, by issuing specialized queries such as $cpu < 80\%$. The *range queries* typically require $O(\log^2(n))$ messages to complete, as explained in Section 5.2.1.1. A meta-molecule *testifies* to the existence of a particular resource molecule; when a node obtains a meta-molecule, it is able to

consequently obtain the original resource molecule the meta-molecule derives from.

5.2.6 Workflow Scheduling Process

Despite the wide variety of existing scheduling heuristics, none of them have been shown to deliver an efficient scheduling algorithm for both types of scientific workflows: data intensive and computation intensive. The selection of an appropriate workflow scheduling algorithm depends of several parameters estimated by users, such as communication costs and task completion times, making the schedulers prone to *judgment errors*. To this respect, the framework proposed in this chapter aims at providing a decentralized and *just-in-time* task-to-resource mapping, on top of which any workflow scheduling heuristic can be implemented. Hence, in our system, workflow scheduling algorithms such as HEFT, LMT, CPOP or DLS, can be supported. A set of these algorithms (Min-min, HEFT, LMT, OLB, and Max-Min) have been implemented using HOCL and are available on-line [7] and also in the Appendix A.3.

5.2.6.1 Inter-layer Execution Model

The execution of rules is event-driven: a rule is triggered when a node receives a molecule or a workflow. Three entities can provoke a rule's execution: a workflow, a level molecule and a task molecule. Upon the receipt of a workflow or a level molecule, a node locally triggers rules to decompose the workflow into levels or levels into tasks, respectively. Afterwards, the levels and tasks generated are hashed and stored in the DHT.

On the other hand, when a node receives a task molecule, it has to find a resource to map it to. Thus, the node constructs a range query in which it lists the requirements a resource has to satisfy and then lets the DHT's range search mechanism (second layer) find it a match. If a matching resource meta-molecule has been found, the node proceeds to the next step: retrieving the resource molecule itself containing information for the actual execution of the task. Finally, the node sends its task to the resource node for execution.

5.2.6.2 Workflow Scheduling Flow

Workflows received by nodes are described using the *chemical workflow definition*. The general shape of this workflow representation is as follows: the main solution is composed of as many *task molecules* as there are tasks (or services) participating in the workflow. Each sub-solution represents a task with its own data and control dependencies with other tasks. This chemical representation of workflows has been shown to be appropriate to express the decentralized execution of a wide variety of workflows patterns [FTP11].

[7] http://www.irisa.fr/myriads/members/hfernand/thesisSources/workflowScheduling

Algorithm 14 Chemical workflow representation.

```
17.01  ⟨ // Multiset (Solution)
17.02      TASK : 1 : ⟨serv_1 : res_desc_1⟩ : ⟨DEST : 2, DEST : 3, ...⟩, // TASK_1 definition
17.03      TASK : 2 : ⟨serv_2 : res_desc_2⟩ : ⟨DEST : 4, ...⟩,
17.04      TASK : 3 : ⟨serv_3 : res_desc_3⟩ : ⟨DEST : 4, ...⟩,
17.05      TASK : 4 : ⟨serv_4 : res_desc_4⟩ : ⟨⟩
17.06  ⟩
```

According to this representation, a chemical node, the entry point, receives the workflow and decomposes it in levels, producing level molecules. This step is based on the first phase of algorithms such LMT or HEFT.

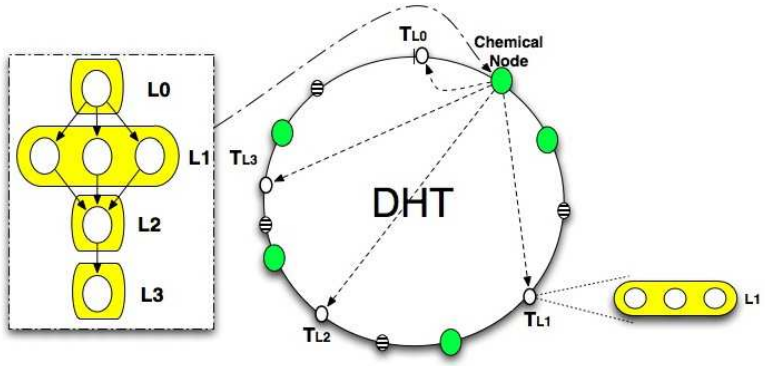

Figure 5.6: Workflow decomposition.

Once the workflow has been received, it triggers the *workflowDecomp* rule which assigns reorganizes the task molecules into levels. This rule consumes the molecules representing the different tasks in the workflow, and produces level molecules, one per level. In Algorithm 15, we present a simplified version of the *workflowDecomp* rule for the workflow decomposition. However, a more detailed version of the HOCL program in charge of this operation can be depicted in Appendix A.3.1.

Algorithm 15 Generic rules.

```
                        — WORKFLOW DECOMPOSITION —
18.01  let workflowDecomp = replace ⟨ Task1:⟨ ?ω1 ⟩, Task2:⟨ ?ω2 ⟩, ..., Taskn:⟨ ?ωn ⟩ ⟩
18.02                       by Level:1:⟨ Task1 ⟩, Level:2:⟨ Task2 ⟩, Level:N:⟨ Taskn ⟩

                        — LEVEL DECOMPOSITION —
18.03  let levelDecomp = replace-one Level:num:SCHEDULED:⟨ Task1, ... ,Taskn ⟩
18.04                       by Task1, ... ,Taskn

                        — TASK TO RESOURCE MAPPING —
18.05  let mapTaskRes = replace Task:idTask:task_res, Res:idRes:⟨ feature1, ... ,featuren ⟩
18.06                       by system.deploy( idTask, idRes )
18.07                       if (Task.isCompatibleWith(Res))
```

Next, we explain the way in which the system couples tasks and resources. Tasks have to be scheduled level by level, calling for coordination between nodes involved in scheduling a given workflow, in order to allow the system to pass from scheduling tasks of one level to the next one, and so on until reaching the last level and delivering the final result to the requesting user. More precisely, the system has to ensure tasks from level i are not being scheduled for execution before or during the scheduling of tasks from level $i-1$. To distinguish between levels which can be scheduled and those which have to wait on scheduling we use two types of molecules: Level : *num* : *READY* and Level : *num*, respectively. The initial workflow decomposition through the activation of the rule *workflowDecomp* produces only Level : *num* molecules to indicate that none of the levels can be scheduled at that time. However, as the scheduling goes on, these molecules will, one by one, turn into Level : *num* : *READY* molecules, indicating that the tasks of the previous levels have been completed and that the tasks of the next level can be scheduled for execution.

To extract tasks from level molecules, a node uses the *levelDecomp* rule. This rule consumes a level molecule with its state set to *READY*, and produces as many task molecules as there are tasks in the given level (Algorithm 15 on Line 18.03). The task molecules are then hashed and stored in the DHT.

Upon receipt of a task molecule, a node uses the *mapTaskRes* rule to map the task to the best suitable resource it can find (Algorithm 15 on Line 18.05). Using the resources' requirements indicated in the task molecule, the second DHT layer is scanned by a *range query* reflecting the **if**-clause of the *mapTaskRes* rule. If a matching resource molecule is found, the rule produces a molecule that deploys the given task onto the resource found during molecule capture (denoted by the special *system.deploy()* molecule in Algorithm 15). Note that, the resource molecule is consumed in the reaction, preventing other tasks to be scheduled before it has been republished. Once a node receives the notification from the system that its task has been completed, it notifies the node holding the level molecule for this task, which, in turn, keeps track of completed tasks. When all of its tasks have been completed, the node holding the (currently active) level molecule retrieves the inactive molecule of the next level. It then deletes it and creates a new, active level molecule and stores it with the same hash identifier. This act allows the next

level to be decomposed and its tasks to be scheduled. This process carries on until the tasks of the last level have finished their execution. At the end, its node collects all of the results and transfers them to the entry node, which delivers them to the client which submitted the workflow for execution.

Multiple Workflow Scheduling Example. Let us consider two workflows, A and B, where task molecules of level *L1* (in both workflows) are awaiting scheduling, as illustrated on Figure 5.7. Task molecules A, B and C from *workflowA* and D and E from *workflowB* are ready to be executed. There are four available resources. This leads to a situation where not all of the tasks can be scheduled concurrently. Each of the five nodes holding task molecules tries to grab a matching resource molecule (due to the execution of the *mapTaskRes* rule). Supposing the nodes holding B, C, and D successfully grab their respective resource molecules, they execute a reaction following the *mapTaskRes* rule by deploying their tasks onto the matching resources. If the only available resource left does not meet the needs of neither A nor E, the nodes holding them wait for a small predefined amount of time and retry fetching a resource molecule. Each of the nodes repeats this cycle until it is able to grab the desired resource molecule and, thus, execute the task it holds.

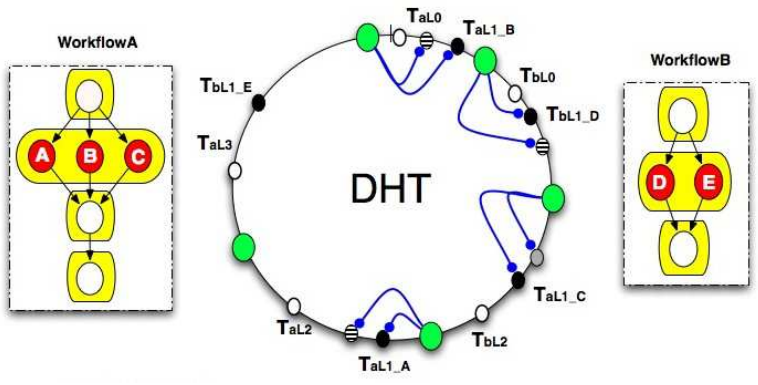

Figure 5.7: Multiple workflow mapping.

5.3 Evaluation

5.3.1 Simulation Set-up

To better capture the behavior and expected performance of the proposed system, a Python-based simulator was built. It simulates a two-layered, DHT-structured network of nodes offering storage and computing power. The nodes also store the meta-information about available resources (the meta-molecules). Workflows sent to this network are processed and scheduled following the decentralized coordination model described in Section 5.2. The simulator operates in discrete time steps.

5.3.1.1 Goals and Assumptions

Our goal is to generally show the overhead, in terms of latency and network load, of the scheduling process itself, as we do not intend to provide a new scheduling algorithm, but

a framework for decentralized coordination for large-scale scheduling. According to that, some assumptions were made. A workflow's depth (the number of levels) was randomly chosen between 3 and 10, with each level containing an arbitrarily assigned number of tasks (between 1 and 15). Furthermore, a task's duration was voluntarily kept low (between 1 and 10 time steps) as it facilitates the evaluation of the framework's overhead. Finally, the computing power of nodes and the capacity of links in the network were virtually unbounded. While such an assumption could sound unrealistic, our simulation did not intend to provide real accuracy in actual settings, as only a real-world deployment could do that. Our validation is oriented towards providing insight into the scalability of the framework.

5.3.2 Results

5.3.2.1 Execution Time

We first tried to capture the scalability in terms of time overhead, when both the number of nodes and workflows increase. Results are depicted in Figure 5.8. The first conclusion is that increasing the number of nodes has an impact on the time taken to schedule workflows. However, this overhead is limited, since the cost of the routing process grows logarithmically with the number of nodes (except for resource retrieval which requires $\log^2(n)$ messages to deal with range queries). Another conclusion that can be drawn by looking at all of the curves together is that when the number of workflows increases, the time to solve them does not increase much. This is a consequence of fully decentralizing the scheduling process, which enables a high degree of parallelism. However, one can argue that, as Figure 5.8 suggests, having fewer nodes leads to better performance. This anomaly is inherent to the assumption that the network links and the computing power are unbounded. Hence, we need to have a look on the network overhead, which is depicted in Figures 5.9 and 5.10.

5.3.2.2 Network Overhead

The series of curves on Figure 5.9 shows the total number of messages generated in the same experiments as earlier. They first suggest that, due to the usage of logarithmic routing, increasing the number of nodes does not impact significantly the network. The curves also show that the number of messages increases with the number of workflows. In fact, the number of messages is directly proportional to the number of tasks to be scheduled, as the scheduling of a task relies on resource retrieval, which needs $O(log^2(n))$ messages to complete. However, this is partly inevitable as each task must be scheduled independently of others. Finally, Figure 5.10 depicts the evolution of the traffic load perceived by each node in the same conditions. While the traffic costs per node increase with the number of workflows, it is also drastically reduced when more nodes take part in the scheduling process. This behavior is a highly desirable one, as we target large-scale platforms. This reinforces the scalability of the whole platform, since the system is able to spread the network load evenly as it increases in size.

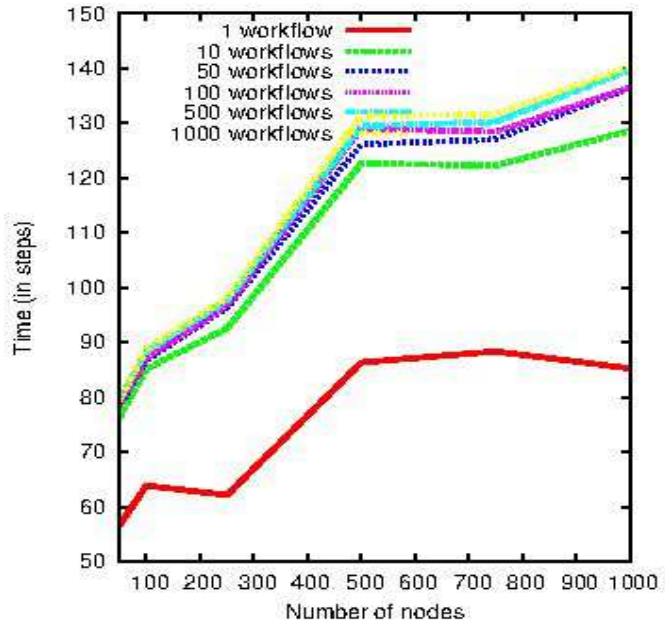

Figure 5.8: Execution time.

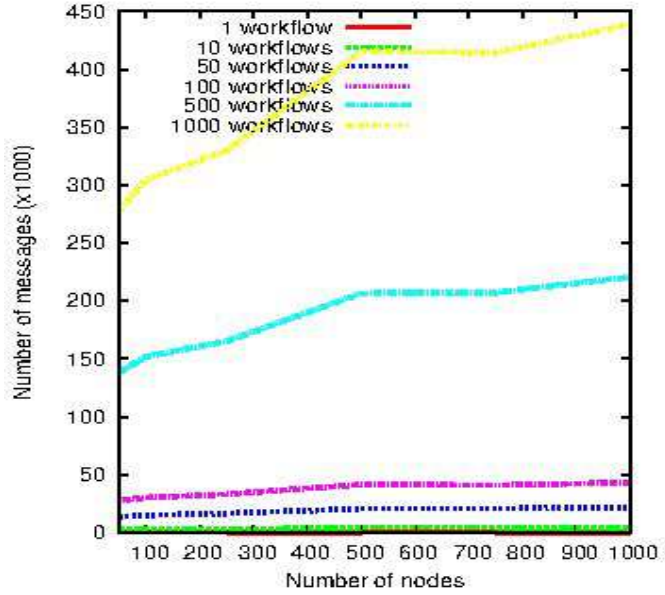

Figure 5.9: Network traffic.

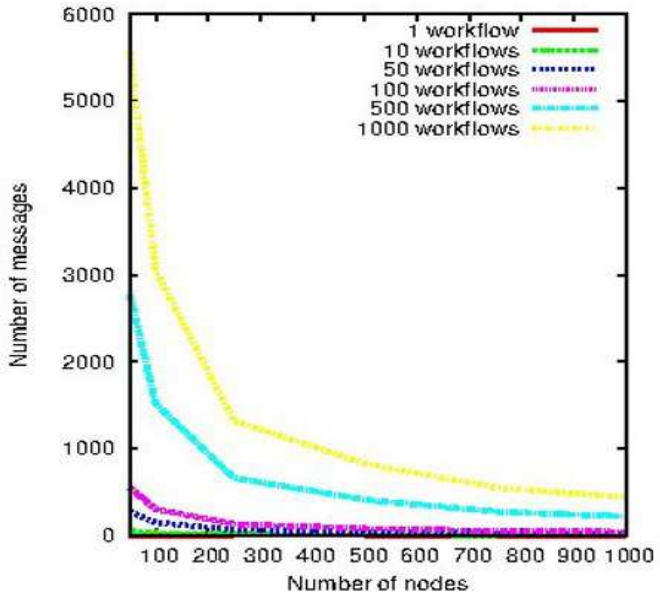

Figure 5.10: Traffic per node.

5.4 Related Works

To the best of our knowledge, there are no related works dealing with the coordination of schedulers in cloud systems. We are, thus, here briefly reviewing works focused on scheduling coordination in federated grids. Note that, due to the nature of grid systems, these works do not address elasticity.

There is a large number of centralized approaches dealing with interoperability drawbacks of federated Grids. Recently, a higher level brokering service was proposed to solve this interoperability issues [81], authors build a meta-broker on top of the resource managers from different Grids and uses the obtained meta-data information to decide where to schedule independent jobs. This approach extracts meta-data from the different resource management systems using technologies of the semantic Web area.

Despite the existence of these centralized approaches, we consider centralized scheduling systems more appropriated to map resources for a single Grid than for scheduling federated Grids. Thus, several models have been proposed to enable a decentralized scheduling in federated Grids [56, 83, 108, 111, 140].

In [108, 140], a decentralized and dynamic resource allocation mechanism was proposed, agents/schedulers dynamically search for the more efficient resource across Grids through negotiation. Therefore, the searching process finishes when the scheduler of a Grid finds the appropriate resource. This type of scheduling is known as *hierarchical*. This architecture can suffer from potential single points of failure: when the scheduler of a Grid fails, none of its resources can be used. In contrast with these works, in our approach all the nodes in the overlay participate to the scheduling process, avoiding possible point of failure.

More recently, in work [56], the authors motivate the need of interlinking Grid sys-

tems through peering arrangements to enable resource sharing. A component known as *InterGrid Gateway* is in charge of the management of its own Grid and the negotiation with the others InterGrid Gateway's from different Grids. An application requests some resources to its Grid resource manager, if it is not able to provide them, then it forwards part of the request to the Gateway which mediates with the other Gateways to select the best resources. This work identifies the key problems as functionals as economical in realizing a federation of Grids. In the same vein, a decentralized model was also proposed in [83], where a meta-scheduler is built on top of each grid infrastructure of a federation, implementing the appropriate scheduling algorithm. This approach uses GridWay as a meta-scheduler which has the whole information about the entire federation. Both approaches [56, 83] present some similarities since for a request of N resources to a Grid, it will face the request if there are N available resources, otherwise additional resources will be requested to other Grids in the federation. Any ranked resource in our chemical overlay is available to be mapped (independently of the distributed infrastructure to which belongs).

There are several works proposing peer-to-peer based solutions to the Grid scheduling problems [66, 111, 133]. Ranjan, Rahman and Buyya [111] designed a P2P coordination space where workflow schedulers cooperate among them enabling a fully decentralized scheduling. This approach implements a DHT representing a distributed blackboard split in regions. Each region is managed by one Grid peer. Thus, the resource providers post their requests into this blackboard, and depending of the region in which the request falls, the broker/agent searchs for the suitable resource in its Grid. This work provides a fully decentralized scheduling model, each Grid in the federation has one broker enabling a point of failure. Our decentralized architecture shares some similarities with this approach. However in our system any resource in the community can schedule tasks.

In works [66, 133], the authors rely on unstructured P2P overlays as communication and coordination system to schedule computation-intensive jobs on federated Grids. Like in our work, there are no predefined schedulers, all the nodes can schedule jobs by propagating a request across the entire set of nodes using a Gossip-based protocol. At the end, the node which transferred the request will select the more appropriate resource from the candidates. Instead of propagating a request across all the neighbours, the chemical system ranks the resources into the DHT allowing to find the best candidate in a short period of time.

In contrast to these approaches, our framework intends not only to decentralize the scheduling process, but also to maximize the efficiency of scheduling algorithms to be deployed, as the coordination layer, built upon a structured network globally ranking resources, enables a global knowledge of available resources in the platform, ensuring that each task will be run on an adequately and accurately chosen resource.

5.5 Conclusion

The evolution of clouds towards community clouds raises the need for large-scale, coordinated workflow mechanisms where the whole set of resources and jobs are associated in a transparent way. This calls for new mechanisms for large-scale coordination mechanisms. This chapter proposes a fully decentralized coordination space relying on a chemical metaphor: workflows, jobs and resources are molecules to be consumed, *i.e.*,

matched. The underlying communication layer ensures a fully decentralized execution of this matching, by relying on a DHT. Moreover, the *DHT-driven* coordination enables a *just-in-time* scheduling technique capable of matching a task to its currently perfect resource candidate. Simulations were conducted, establishing further the feasibility and scalability of the approach.

This chapter has two main features in comparison with the previous work. First, the system is here resource-aware by taking into account the selection of the more appropriate resources when processing workflows. Secondly, the workflow scheduling process is fully decentralized among all the nodes in the community. The multiset is now implemented through a DHT. In such a way, it is planned to extend the *HOCL-TS* prototype presented in Chapter 4 with these functionalities.

Conclusion

With the proliferation of Web services and the emergence of new service infrastructures, service-based applications represent an important driver for the present and future of software design. However, even though the existing service coordination models have shown their adequacy dealing with these service-based applications, they suffer from centralization, lack of dynamicity and low level of abstraction, which limit their reliability and adoption in emerging service platforms. Consequently, there is a demand for more dynamic, loosely coupled and high level coordination models. In this thesis, we proposed a decentralized workflow system based on an unconventional coordination model that relies on the chemical metaphor. While the coordination is decentralized, loosely coupled and distributed among chemical workflow engines, the chemistry-inspired model brings dynamicity, wide expresiveness and a high level of abstraction for the execution of service compositions. In the same idea, we take the resources into account in our system to improve the workflow execution on distributed infrastructures. Thus, we designed a workflow scheduling framework that ensures a just-in-time and a fully decentralized workflow-resources matching by relying on a chemically coordinated shared space, a multiset built on top of a DHT.

In Chapter 1, we presented the main methodologies for service composition, as well as the more mature and relevant systems for the workflow management. However, the majority of these approaches are still mostly based on a centralized coordination model. Their workflow executable languages are intrinsically static and do not provide concepts for autonomous and decentralized workflow execution. As a consequence, such systems present several drawbacks, mainly dealing with scalability, dynamicity, fault tolerance, and security.

In order to tackle these issues, it has become crucial to propose flexible and decentralized coordination mechanisms. Thus, Chapter 2 presented some of the more promising flexible model for service coordination, which basically provide dynamic and loosely coupled interaction mechanisms among services. In particular, the chemical coordination model was shown as a promising paradigm naturally capturing parallelism, distribution, dynamics, as well as a high level of abstraction. This model, as a conjunction of a rule-based and a tuplespace-based model, was applied to coordinate workflows in this thesis.

In Chapter 3 and Chapter 4, we presented the design and the implementation of a decentralized chemically coordinated workflow management system. This system was based on a shared multiset containing the information on both data and control dependencies needed for coordination, and where chemical engines were co-responsible for carrying out the execution of a workflow. In Chapter 3, we have proposed concepts for a chemistry-inspired autonomic workflow execution, namely *molecular composition*. Such

an analogy was shown to be able to express the decentralized and autonomous execution of a wide variety of workflow patterns by composing reactions (reactions trigger another reactions). Although the chemical language is naturally data-driven, the modelling of control-driven structures is also supported, as highlighted in Chapter 3. Thus, a ready-to-use HOCL library for this purpose has been designed and experimented, showing the wide expressiveness of the chemical model. Consequently, in Chapter 4, the chemistry-inspired workflow management system was prototyped based on the HOCL language, for a centralized, tuplespace-based and fully decentralized (the multiset is statically pre-distributed across nodes) using real workflows, providing a proof of concept and suggesting promising performance in comparison to current reference workflow management systems.

The experiments conducted using these prototypes allowed to show the benefits of the decentralization when processing computation and data intensive workflows, as the coordination workload was decentralized and distributed among chemical engines.

While the advantages of this model are well-established, the results presented in this thesis open the doors to the future consideration of this unconventional model for service coordination.

Finally, in Chapter 5, we took resources of the platform into account to complete this chemistry-inspired workflow system. Thereby, we proposed a fully decentralized workflow scheduling framework as a solution to the current drawbacks when scheduling workflows in *community clouds* platforms. Note that, a fully decentralized implementation of the multiset is achieved due to the use of a DHT. This two-layer framework provides support for just-in-time multiple workflow scheduling. Based on a chemically-coordinated shared space, all the machines of a community work as schedulers being able to map any available resource in a chemical overlay following simple chemical rules. Consequently, a campaign of simulations were conducted on the proposed system to evaluate its behavior showing promising results regarding the feasibility and scalability of this approach.

Future Works

Even though our chemistry-inspired workflow management system covers an essential part of a workflow life cycle, some open issues should be tackled in future works.

- *Autonomous transformation.* To facilitate the adoption of the chemical model, a mechanism to translate traditional workflow definitions into chemical programs is needed. Languages such as BPEL or SCUFL are commonly used in industry and scientific domains. Their graphical and user-friendly tools facilitate the design of workflows by end-users. Thus, a module could be plugged on top of our workflow system to automatically transform a BPEL, SCUFL or other proprietary specifications into a chemical workflow. The resulting chemical workflow definition would be later processed in a decentralized manner. The basis of a transformation model have already been established in Appendix A.1, however further investigations have to be conducted.

- *Workflow partitioning.* It is a technique to transform centric-based workflow definitions into self-describing portions of workflow which will be executed in a decentralized manner. In the current version of our system, the workflow partitioning

basically associates one partition per Web service involved in a workflow. However, many other criteria may be considered when partitioning a workflow definition, as suggested in [64]. The role of a service in a specification, its owner organization, its physical location, the location of the data to be consumed, as well as other QoS and distributed constraints are some of the characteristics that should be taken into account when partitioning the service interactions of a workflow. This could constitute an interesting topic by selecting the characteristics that improve the performance of a specific workflow, *i.e.*, the data location for data-intensive workflows.

- *QoS assurance.* This technique selects one service to participate in a workflow among a set of available services in order to accomplish a specific QoS contract. To do that, QoS systems use *dynamic binding*, allowing to select at runtime the service which better satisfies some QoS conditions. Some investigations should be conducted to exploit the dynamic and autonomous behavior of the chemical coordination model to ensure *QoS* constraints of a service composition, as envisioned in [98]. Indeed, our chemistry-inspired workflow system includes the *Service caller* component that enables the run time selection of services by only using its WSDL file descriptor, as explained in Chapter 4. The *Service caller* component could ensure the accomplishment of some QoS conditions by plugging a module in charge of this verification.

- *Distributing the multiset.* It is another interesting subject on which more experiments should be conducted, in particular, to evaluate the fault tolerance and persistence properties when processing workflows in a fully decentralized manner. Chapter 5 presented a fully decentralized multiset for workflow scheduling that shows the benefits by distributing this space. In particular, a software prototype is being developed based on the two-layer architecture introduced in Chapter 5.

- *Task or service clustering.* This scheduling technique is used to minimize the completion time of a workflow in a distributed infrastructure. It increases the computational granularity of workflow tasks, by grouping fine-grained tasks into the same resource for the execution. This mechanism may be further studied and included to our workflow system. The current version of our approach maps one task (service) onto one resource, which may waste money and energy, in terms of a distributed infrastructure. The task clustering improves the performance, as fine-grained tasks are grouped and deployed on less resources instead of one per task. This mechanism has been included in Pegasus showing the benefits by using this technique [54]. However, additional studies should be carried to determine how to group tasks depending on the workflow structure, the target execution platform and QoS constraints.

The contributions presented in this thesis can be applied to solve problems in many other areas. For instance, our chemistry-inspired coordination model has been successfully applied as a proof of concept to show the benefits of Agile Service Networks (networks of services) in the context of Global Software Engineering (GSE) [TFRZ+12].

Finally, the emergence of new service infrastructures, where a high level of abstraction, elasticity, and dynamic adaptation are mandatory requirements, led to a high demand

for new models being able to represent both service collaborations and platforms, as well as their inherent characteristics. In such a way, the chemical metaphor arises up as a source of inspiration for the development of unconventional coordination models by providing a high level of abstraction and a flexible mechanism for the execution of service compositions.

Bibliography

[1] SOAP Specification. http://www.w3.org/TR/soap/, 2000.

[2] W3C Web Service Descriptor Language 1.1. http://www.w3.org/TR/wsdl, 2001.

[3] WSFL -Web Service Flow Language. http://xml.coverpages.org/wsfl.html, 2001.

[4] Bittorrent. http://bittorrent.com, 2003.

[5] OASIS UDDI Specification TC. http://www.oasis-open.org/, 2005.

[6] GWENDIA Workflow Language – Proposal. http://gwendia.i3s.unice.fr, 2009.

[7] The Venus-C European Project. http://www.venus-c.eu/Pages/Home.aspx, 2010.

[8] Grid'5000. http://www.grid5000.fr, June 2011.

[9] The Magellan Research Project. http://magellan.alcf.anl.gov/, June 2011.

[10] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt, and Jie Wu. Advanced Peer-to-Peer networking: The P-Grid system and its applications. *PIK - Journal, Special Issue on P2P Systems*, 26(2):86–89, 2003.

[11] Lifeng Ai, Maolin Tang, and Colin Fidge. Partitioning composite web services for decentralized execution using a genetic algorithm. *Future Gener. Comput. Syst.*, 27(2):157–172, 2011.

[12] Phillipa Oaks Alistair Barros, Marlon Dumas. A critical overview of the web services choreography (WS-CDL). *BP-Trends*, 2005.

[13] Gustavo Alonso, C. Mohan, Divyakant Agrawal, and Amr El Abbadi. Functionality and limitations of current workflow management systems. *IEEE Expert*, 12, 1997.

[14] Martin Alt and Sergei Gorlatch. Using Skeletons in a Java-based Grid system. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 682–693. Springer-Verlag, 2003.

[15] Martin Alt, Sergei Gorlatch, Andreas Hoheisel, and Hans werner Pohl. A grid workflow language using high-level petri nets. In *Parallel processing and applied mathematics, 6th International Conference, PPAM 2005*, volume 3911, pages 715–722, Pozna , Poland, 2005. Springer Berlin.

[16] Kaizar Amin, Gregor von Laszewski, Mihael Hategan, J. Nestor Zaluzec, Shawn Hampton, and Albert Rossi. GridAnt: a client-controllable grid workflow system. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004*. IEEE, 2004.

[17] Assaf Arkin. Business Process Modeling Language. BPMI, 2002.

[18] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.

[19] Vijayalakshmi Atluri, Soon Ae Chun, Ravi Mukkamala, and Pietro Mazzoleni. A decentralized execution model for inter-organizational workflows. *Distrib. Parallel Databases*, 22(1):55–83, 2007.

[20] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Chemical Specification of Autonomic Systems. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering*, 2004.

[21] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Principles of chemical programming. *Electronic Notes in Theoretical Computer Science*, 124(1):133–147, 2005.

[22] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Generalised multisets for chemical programming. *Mathematical Structures in Computer Science*, 16(4):557–580, 2006.

[23] Jean-Pierre Banâtre, Nicolas Le Scouarnec, Thierry Priol, and Yann Radenac. Towards Chemical Desktop Grids. In *IEEE International Conference on e-Science and Grid Computing*, pages 135–142. IEEE, 2007.

[24] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Sci. Comput. Program.*, 15(1):55–77, 1990.

[25] Jean-Pierre Banâtre, Christine Morin, and Thierry Priol. Fault tolerant autonomic computing systems in a chemical setting. In *Dependable and Historic Computing*, volume 6875, pages 118–129. 2011.

[26] Jean-Pierre Banâtre, Thierry Priol, and Yann Radenac. Service orchestration using the chemical metaphor. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 79–89. 2008.

[27] Jean-Pierre Banâtre, Thierry Priol, and Yann Radenac. Chemical Programming of Future Service-oriented Architectures. *Journal of Software*, 4(7):738–746, 2009.

[28] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards distributed bpel orchestrations. *ECEASST*, 3, 2006.

[29] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, pages 1–10, 2009.

[30] Adam Barker, Christopher D Walton, and David Robertson. Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009.

[31] Umesh Bellur and Siddharth Bondre. xSpace: a tuple space for XML & its application in orchestration of web services. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 766–772, New York, NY, USA, 2006. ACM.

[32] G. Bruce Berriman, Ewa Deelman, John Good, Joseph Jacob, Daniel Katz, Carl Kesselman, Anastasia Laity, Thomas Prince, Gurmeet Singh, and Mei hu Su. Montage: A grid enabled engine for delivering custom science-grade mosaics on demand. *In Proceedings of SPIE Conference 5487: Astronomical Telescopes*, 2004.

[33] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[34] Marin Bertier, Marko Obrovac, and Cédric Tedeschi. A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms. In *13th International Conference on Distributed Computing and Networking*, volume 7129, pages 1–15, Hong-Kong, China, January, 3-6 2012.

[35] Martin Bichler and Kwei-Jay Lin. Service-Oriented computing. 39(3):99–101, 2006.

[36] Walter Binder, Ion Constantinescu, and Boi Faltings. Decentralized orchestration of CompositeWeb services. In *Proceedings of the IEEE International Conference on Web Services*, pages 869–876. IEEE Computer Society, 2006.

[37] Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005*, volume 2, pages 759– 767 Vol. 2. IEEE, 2005.

[38] Jeremy Bolie, Michael Cardella, Stany Blanvalet, Matjaz Juric, Sean Carey, Praveen Chandran, Yves Coene, and Kevin Geminiuc. *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development: Ten practical real-world case studies combining business ... management and web services orchestration.* Packt Publishing, July 2006.

[39] Tracy Braun. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[40] Damien Bright and Gerald Quirchmayr. Supporting Web-Based collaboration between virtual enterprise partners. In *15th International Workshop on Database and Expert Systems Applications, 2004. Proceedings*, pages 1029– 1035. IEEE, 2004.

[41] Nat Brown and Charlie Kindel. Distributed component object model protocol — dcom/1.0. Internet Draft, Network working group, 1998.

[42] Paul A. Buhler and Jose M. Vidal. Enacting BPEL4WS specified workflows with multiagent systems. *In Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[43] Manuel Caeiro, Zlot Nemeth, and Thierry Priol. A chemical workflow engine for scientific workflows with dynamicity support. In *Third Workshop on Workflows in Support of Large-Scale Science, 2008. WORKS 2008*, pages 1–10. IEEE, Nov 2008.

[44] Eddy Caron, Frederic Desprez, David Loureiro, and Adrian Muresan. Cloud computing resource management through a grid middleware: A case study with DIET and eucalyptus. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, pages 151–154. IEEE Computer Society, 2009.

[45] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *In Proceedings of the 13th International World Wide Web Conference, (WWW'04)*, pages 134–143, New York, NY, USA, 2004. ACM.

[46] Anis Charfi and Mira Mezini. AO4BPEL: an aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344, 2007.

[47] Jing-Ying Chen. Rewrite rules as service integrators. In *Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Second International Workshop, RuleML 2003*, pages 182–187. Springer Berlin / Heidelberg, 2003.

[48] Jen-Yao Chung, Kwei-Jay Lin, and Richard G. Mathieu. Guest editors' introduction: Web services Computin–Advancing software interoperability. *IEEE Computer*, 36(10):35–37, 2003.

[49] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *Proceedings of the 27th International Conference on Software Engineering, (ICSE '05)*, pages 69–77. ACM, 2005.

[50] Krysten Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *The Eighth International Symposium on High Performance Distributed Computing, 1999. Proceedings*, pages 219–228. IEEE, 1999.

[51] Gero Decker, Margarit Kirov, J M Zaha, and Marlon Dumas. Maestro for let's dance: An environment for modeling service interactions. *BPM Demo Session 2006*, page 32, 2006.

[52] Gero Decker, Oliver Kopp, Frank Leymann, Kerstin Pfitzner, and Mathias Weske. Modeling service choreographies using BPMN and BPEL4Chor. In *Advanced Information Systems Engineering*, pages 79–93. 2008.

[53] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: extending BPEL for modeling choreographies. *IEEE International Conference on Web Services*, 0:296–303, 2007.

[54] Ewa Deelman, Gaurang Mehta, Gurmeet Singh, Mei-Hui Su, and Karan Vahi. Pegasus: Mapping large-scale workflows to distributed resources. In *Workflows for e-Science*, pages 376–394. Springer London, 2007.

[55] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[56] Marcos Dias de Assunção, Rajkumar Buyya, and Srikumar Venugopal. Intergrid: a case for internetworking islands of grids. *Concurr. Comput. : Pract. Exper.*, 20:997–1024, June 2008.

[57] Patrick Downes, Oisín Curran, John Cunniffe, and Andy Shearer. Distributed radiotherapy simulation with the webcom workflow system. *International Journal of High Performance Computing Applications*, 24:213–227, 2010.

[58] Troy Bryan Downing. *Java RMI: Remote Method Invocation.* IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[59] Johan Eker, W. Jörn Janneck, A. Edward Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[60] Lauren Erwin and Jones Bob. Enabling Grids for e-Science: EGEE Project. 2008.

[61] Thomas Fahringer, Radu Prodan, Rubing Duan, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and Marek Wieczorek. ASKALON: a grid application development and computing environment. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 122–131. IEEE Computer Society, 2005.

[62] Thomas Fahringer, Jun Qin, and Stefan Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. In *IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005.*, volume 2, pages 676 – 685 Vol. 2, 2005.

[63] Walid Fdhila and Claude Godart. Toward synchronization between decentralized orchestrations of composite web services. In *5th International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009*, pages 1–10. IEEE, 2009.

[64] Walid Fdhila, Ustun Yildiz, and Claude Godart. A flexible approach for automatic process decentralization using dependency tables. In *IEEE International Conference on Web Services, 2009. ICWS 2009*, pages 847–855. IEEE, 2009.

[65] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[66] Marco Fiscato, Paolo Costa, and Guillaume Pierre. On the feasibility of decentralized grid scheduling. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:225–229, 2008.

[67] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. *In 2nd International Workshop On Peer-To-Peer Systems (IPTPS'03*, pages 118—128, 2003.

[68] Ian Foster and Carl Kesselman. The globus toolkit. pages 259–278, 1999.

[69] Pi4 Technologies Foundation. Pi4SOA choreography system. 2005.

[70] Lars Fredlund. Implementing WS-CDL. Universidade de Santiago de Compostela, 2006. In proceedings of JSWEB 2006 (II Jornadas Científico-Técnicas en Servicios Web).

[71] Jaime Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: a computation management agent for multi-institutional grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001.*, pages 55–63, 2001.

[72] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.

[73] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of Data-Intensive applications on grids with MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347–360, 2008.

[74] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference - Volume 17*, pages 191–200, Adelaide, Australia, 2003. Australian Computer Society, Inc.

[75] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34:631–651, June 2004.

[76] David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Alistair Barros, Marlon Dumas, and Arthur H. M. Hofstede. Service interaction patterns. In *Business Process Management*, volume 3649, pages 302–318. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[77] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the Association for Computing Machinery, ACM*, 24(2):280–289, 1977.

[78] Michael A Iverson, Füsun Özgüner, and Gregory J Follen. Parallelizing existing applications in a distributed heterogeneous environment. *In 4th Heterogeneous Computing Workshop (HCW)'95*, pages 93—100, 1995.

[79] Kurt Jensen and Grzegorz Rozenberg, editors. *High-level Petri Nets: Theory and Application.* Springer, 1 edition, 1991.

[80] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[81] Attila Kertesz, Peter K Kacsuk, Ivan Rodero, Francesc Guim, and Julita Corbalan. Meta-Brokering requirements and research directions in state-of-the-art grid resource management. *CoreGRID Technical Report*, 9(4):181–188, 2007.

[82] Zakir Laliwala, Rahul Khosla, Pritha Majumdar, and Sanjay Chaudhary. Semantic and rules based Event-Driven dynamic web services composition for automation of business processes. In *Services Computing Workshops, 2006. SCW '06. IEEE*, pages 175–182, 2006.

[83] Katia Leal, Eduardo Huedo, and Ignacio M. Llorente. A decentralized model for scheduling independent tasks in federated grids. *Future Generation Computer Systems*, 25:840–852, 2009.

[84] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Daios: Efficient dynamic web service invocation. *IEEE Internet Computing*, 13(3):72–80, 2009.

[85] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 487–491, New York, NY, USA, 2004. ACM.

[86] Bertram Ludascher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1039–1065, August 2006.

[87] Alexandros Marinos and Gerard Briscoe. Community cloud computing. In *CloudCom*, pages 472–484, 2009.

[88] Daniel Martin, Daniel Wutke, and Frank Leymann. A novel approach to decentralized workflow enactment. In *Enterprise Distributed Object Computing Conference, IEEE International*, pages 127–136, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

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

[90] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18(2):215–224, 1989.

[91] Sun Meng and Farhad Arbab. Web services choreography and orchestration in reo and constraint automata. In *Proceedings of the 2007 ACM symposium on Applied computing SAC'07*, pages 346–353, Seoul, Korea, 2007.

[92] Huhns N. Michael and Singh P. Mahendra. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, 2005.

[93] Rosa Anna Micillo, Salvatore Venticinque, Nicola Mazzocca, and Rocco Aversa. An agent-based approach for distributed execution of composite web services. In *IEEE International Workshops on Enabling Technologies*, pages 18–23, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[94] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *Internet Computing, IEEE*, 8(6):51–59, 2004.

[95] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[96] Frederic Montagut and Refik Molva. Enabling pervasive execution of workflows. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, page 10 pp., 2005.

[97] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *Proceedings of the 19th conference on object-oriented programming, systems, languages, and applications*, pages 170–187. ACM, 2004.

[98] Claudia Di Napoli, Maurizio Giordano, Jean-Louis Pazat, and Chen Wang. A Chemical Based Middleware for Workflow Instantiation and Execution. In *ServiceWave*, pages 100–111, 2010.

[99] Zlot Nemeth, Christian Perez, and Thierry Priol. Distributed workflow coordination: molecules and reactions. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*. IEEE, 2006.

[100] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. KLAIM: a kernel language for agents interaction and mobility. *IEEE Transactions On Software Engineering*, 24, 1997.

[101] OASIS. Web services business process execution language, (WS-BPEL), Version 2.0, 2007.

[102] Marko Obrovac and Cédric Tedeschi. When Distributed Hash Tables Meet Chemical Programming for Autonomic Computing. In *15th International Workshop on Nature Inspired Distributed Computing (NIDisC 2012), in conjunction with IPDPS 2012*, Shanghai, China, May, 21-25 2012. IEEE. To appear.

[103] Rober Lucchi University of Bologna Gianluigi Zavattaro University of Bologna. WSSecSpaces: a secure Data-Driven coordination service for web services applications. *SAC'04, Nicosia Cyprus, ACM*, March 2004.

[104] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1067–1100, August 2006.

[105] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[106] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[107] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[108] Florin Pop, Ciprian Dobre, and Valentin Cristea. Decentralized dynamic resource allocation for workflows in grid environments. In *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC '08*, pages 557–563. IEEE, 2008.

[109] Jelica Protić, Milo Tomasević, and Veljko Milutinović. *Distributed shared memory.* John Wiley and Sons, 1998.

[110] Yann Radenac. *Programmation "chimique" d'ordre supérieur.* Thèse de doctorat, Université de Rennes 1, April 2007.

[111] Rajiv Ranjan, Mustafizur Rahman, and Rajkumar Buyya. A decentralized and cooperative workflow scheduling algorithm. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID '08*, pages 1–8. IEEE, 2008.

[112] Jan C. Recker. BPMN modeling – who, where, how and why. *BP-Trends*, 2008.

[113] Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. *Applied SOA: Service-Oriented Architecture and Design Strategies.* Wiley, 2008.

[114] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[115] Nick Russell, Arthur Hofstede, Wil van der Aalst, and Nataliya Mulyar. Workflow Control-Flow patterns: A revised view. Technical report, 2006.

[116] Rizos Sakellariou, Henan Zhao, Ewa Deelman, Frédéric Desprez, Vladimir Getov, Thierry Priol, and Ramin Yahyapour. Mapping workflows on grid resources: Experiments with the montage workflow. In *Grids, P2P and Services Computing*, pages 119–132. Springer US, 2010.

[117] Schmidt and M. Parashar. Squid: Enabling search in DHT-based systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, 2008.

[118] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-j Schek. Scalable peer-to-peer process management - the OSIRIS approach. *In proceedings of the 2nd International Conference on Web Services (ICWS'2004)*, pages 26—34, 2004.

[119] Jon Siegel. *CORBA fundamentals and programming.* Wiley, 1996.

[120] C. Gilbert Sih and A. Edward Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[121] Mirko Sonntag, Katharina Gorlach, Dimka Karastoyanova, Frank Leymann, and Michael Reiter. Process space-based scientific workflow enactment. *International Journal of Business Process Integration and Management*, 5(1):32 – 44, 2010.

[122] Biplav Srivastava and Jana Koehler. Web service composition - current solutions and open problems. *In ICAPS 2003 Workshop on Planning for Web Services*, pages 28—35, 2003.

[123] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[124] Daniel Stutzbach and Reza Rejaie. Characterizing the two-tier gnutella topology. In *SIGMETRICS*, pages 402–403, 2005.

[125] Wei Tan, Paolo Missier, Ian Foster, Ravi Madduri, David De Roure, and Carole Goble. A comparison of using taverna and BPEL in building scientific workflows: the case of caGrid. *Concurr. Comput. : Pract. Exper.*, 22(9):1098–1117, 2010.

[126] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, Matthew Shields, and Aleksander Slominski. Adapting BPEL to scientific workflows. In *Workflows for e-Science*, pages 208–226. 2007.

[127] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop (HCW)*, pages 3–14, 1999.

[128] Jos J. M. Trienekens, Jacques J. Bouman, and Mark Van Der Zwan. Specification of service level agreements: Problems, principles and practices. *Software Quality Control*, 12(1):43–57, 2004.

[129] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, pages 441–448. IEEE, 2007.

[130] David Gelernter Yale University. Generative communication in linda. *ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1,*, pages 80–112, 1985.

[131] Wil Van der Aalst and Arthur Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[132] Wil Van Der Aalst, Arthur Ter Hofstede, Bartosz Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[133] Xenofon Vasilakos, Jan Sacha, and Guillaume Pierre. Decentralized As-Soon-As-Possible grid scheduling: A feasibility study. In *2010 Proceedings of 19th Int. Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2010.

[134] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In *Coordination Models and Languages*, volume 5521 of *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin / Heidelberg, 2009.

[135] Mirko Viroli and Franco Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, pages 1–17, 2009.

[136] Chen Wang and Jean-Louis Pazat. Using chemical metaphor to express workflow and service orchestration. In *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 1504–1511. IEEE, 2010.

[137] Workflow Management Coalition (WfMC). The XML Process Definition Language (XPDL) / Version 2.0, 2005.

[138] Daniel Wutke, Daniel Martin, and Frank Leymann. Facilitating complex web service interactions through a tuplespace binding. In *Distributed Applications and Interoperable Systems*, pages 275–280. 2008.

[139] Jun Yan, Yun Yang, and Gitesh Raikundalia. Enacting business processes in a decentralised environment with p2p-based workflow support. In *Advances in Web-Age Information Management*, pages 290–297. 2003.

[140] J Yang, Y Bai, and Y Qiu. A decentralized resource allocation policy in minigrid. *Future Generation Computer Systems*, 23:359–366, 2007.

[141] Jian Yang, Mike P. Papazoglou, Bart Orriens, and Willem-Jan van Heuvel. A rule based approach to the service composition Life-Cycle. page 295. IEEE Computer Society, 2003.

[142] Yun Yang. An architecture and the related mechanisms for web-based global cooperative teamwork support. *Int. Journal of Computing and Informatics*, 24, 2000.

[143] Jian Cao Feilong Tang Lin Chen Yi Wang, Minglu Li and Lei Cao. An ECA-Rule-Based workflow management approach for web services composition. *In 4th Int. Conference on Grid and Cooperative Computing (GCC)'05*, 3795:143–148, 2005.

[144] Weihai Yu. Peer-to-Peer execution of BPEL processes. *CAiSE'07 Forum, Proceedings of the CAiSE'07 Forum at the 19th International Conference on Advanced Information Systems Engineering, Trondheim, Norway, 11-15 June 2007*, (CAiSE Forum), 2007.

[145] Weihai Yu. Consistent and decentralized orchestration of BPEL processes. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1583–1584, Honolulu, Hawaii, 2009. ACM.

[146] Weihai Yu and Jie Yang. Continuation-passing enactment of distributed recoverable workflows. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 475–481, New York, NY, USA, 2007. ACM.

[147] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur Hofstede. Let's dance: A language for service behavior modeling. *On the Move to Meaningful Internet Systems 2006 CoopIS DOA GADA and ODBASE*, 54(6):145–162, 2006.

[148] Sonja Zaplata, Kristian Kottke, Lamersdorf Winfried, and Matthias Meiners. Towards runtime migration of WS-BPEL processes. In *Fifth International Workshop on Engineering Service-Oriented Applications (WESOA'09)*. Springer, 2010.

List of Publications

International Conferences Articles

[FPT10] Héctor Fernández, Thierry Priol, and Cédric Tedeschi. Decentralized Approach for Execution of Composite Web Services using the Chemical Paradigm. In *8th International Conference on Web Services (ICWS 2010)*, pages 139–146, Miami, USA, July 5-10 2010. IEEE.

[FTP11] Héctor Fernández, Cédric Tedeschi, and Thierry Priol. A Chemistry-Inspired workflow management system for scientific applications in clouds. In *7th IEEE International Conference on eScience*, volume 0, pages 39–46, Stockholm, Sweden, 2011. IEEE Computer Society.

International Workshop Articles

[FTPW11] Héctor Fernández, Cédric Tedeschi, and Thierry Priol. Decentralized Workflow Coordination through Molecular Composition. In *7th International Workshop on Engineering Service-Oriented Applications WESOA*, In conjunction with ICSOC. pages 22–32, Paphos, Cyprus, 2011. Springer.

[TFRZ+12] Damian Tamburri, Héctor Fernández, S. Ivan Razo-Zapata and Cédric Tedeschi. Simulating Awareness in Global Software Engineering: a Comparative Analysis of Scrum and Agile Service Networks. In *PESOS 4th International Workshop on Principles of Engineering Service-Oriented Systems*, In conjunction with ICSE. 2012. To appear.

National Conferences Articles

[F11] Héctor Fernández. Exécution décentralisée de workflow par composition moléculaire. *20^e Rencontres francophones du parallelisme (RenPar'20)*, Saint Malo, France, 2011.

Technical Reports

[HT11] Héctor Fernández, Cédric Tedeschi and Thierry Priol. Self-coordination of Workflow Execution Through Molecular Composition. Research Report RR-7610, INRIA, June 2011.

[TRFOT12] Héctor Fernández, Marko Obrovac, Cédric Tedeschi. Decentralised Multiple Workflow Scheduling via a Chemically-coordinated Shared Space. Research Report RR-7925, INRIA, April 2012.

[TRFTP12] Héctor Fernández, Cédric Tedeschi and Thierry Priol. A Chemistry-Inspired Workflow Management System for a Decentralized Workflow Execution. Research Report RR-7924, INRIA, April 2012.

Appendix A

General Concepts

A.1 Elements of a Workflow Specification

Let us consider a workflow whose data and control dependencies of the workflow were previously defined at build-time using a traditional workflow definition language, such as the well-known BPEL [101]. However, any workflow definition language could be used for translating one graphical workflow representation into an executable program. We here review several workflow languages and give the equivalences between these languages and an HOCL-based workflow definition that will be executed by chemical engines (HOCL Interpreter).

BPEL is an imperative and control-based workflow language. It includes the explicit definition of the control flow that determines the execution order. Likewise, in BPEL, the Web services are primitive execution blocks, and service composition is achieved using control primitives such as *sequence, parallel, conditionals* and *loops*. In contrast, workflow languages such as Scufl or HOCL are data-driven. Scufl is an XML-based workflow description language. Scufl defines an abstract workflow from a graph of data interactions between different services called *processors*, hiding the complexity of the interoperation of the services to the users. HOCL also presents a data-driven behavior, services are represented as chemical solutions, where data are represented as molecules and computations as the chemical reactions among molecules. HOCL can be also used as a hybrid language (both control and data driven). To provide a control-driven behavior, we need to define additional chemical rules known as *generic*, *i.e.*, independent of any workflow definition. These additional rules, that are part of the HOCL workflow engine, allowing to define the order of execution, as detailed in Section 3.2.4 and Section 3.2.5 of Chapter 3.

Among the variety of existing workflow languages for business domains, such as XPDL [137], BPEL and YAWL; and for scientific domains, such as Kepler, DAX and Scufl. We choose BPEL and Scufl how the most representative of both domains to detail the differences between them and HOCL, as is summarized in the Table A.1.

In Table A.1, Web services definitions are represented in BPEL using <*partnerLinks*> primitive or in Taverna using <*processor*> primitive, while they are represented as ChWSes in HOCL, a ChWS represents one service participating in the workflow.

	BPEL	Scufl/Taverna	HOCL
Web service definition	Defined using <*partnerLinks*>	Defined using <*processor*> plug-ins	Defined using *ChWS*:<...> molecules
Activity definition	Basic and structure activities	Data processing units	Chemical rules
Data definition	Explicit using variables	Implicit (input/output in data units)	Implicit or explicit using molecules
Semantic links	Transfer of control	Transfer of data	Transfer of data & control
Supported patterns	Sequence, parallel split, exclusive-choice, synchronization, simple-merge...	Sequence, parallel split and conditional	Sequence, parallel split, synchronization, simple-merge, exclusive-choice...
Parallel execution	Explicitly defined using <*flow*> primitive	Implicit	Implicit
Workflow management	Centralized (Single coordinator node)	Centralized	Centralized / Decentralized

Table A.1: Comparison of BPEL, Scufl and HOCL.

WS definitions. A BPEL process consists of steps, each step is an *activity*. Activities are a set of primitives like *invoke*, *reply*, *assign*, *flow* among others, which are used for common tasks. In Scufl, activities are data processing units with input/output ports that can be executed as soon as input data are received. Unlike in the chemical paradigm, we use chemical rules to execute these tasks, as summarized in Table A.1.

Data definition. For data definition, in Table A.1, BPEL requires the explicit definition of variables to hold data structures that are meant to be shared among activities. This definition takes additional effort but also brings more flexibility. For example, in BPEL you can define both *global variable* concerning the whole flow and *local variable* whose scope will be a specific activity. In HOCL, molecules within the main solution can be used as global variables without the need for explicit definition thanks to its data-driven behavior, as detailed in Section 3.2.3 of Chapter 3. M, BPEL variables of complex type must be initialized prior to their first use. However, this initialization is not required for the molecules in HOCL and neither in Taverna where the notion of data is directly linked from an output to an input with no initialization.

Data transfer. In function of the information transferred through links among activities or nodes, our system distributes control and data information about the execution. In the scientific workflow area, workflows take the shape of data processing pipelines requiring to express data transfer easily. That is the reason because Taverna is data-driven language. In contrast, in BPEL, control information is only transferred through links representing the order of execution.

Workflow management. In the traditional orchestration model of BPEL, control dependencies and data are managed through a centralized engine, which results in unnecessary data transfer, wasted bandwidth and performance bottleneck during the execution

of workflows. In Taverna, although the language offers a distributed execution, its coordination is still managed by a centralized engine. In contrary, the chemical execution model supports a centralized and decentralized workflow execution. The decentralized workflow system is based on several chemical local engines which are co-responsible of the coordination during the execution, as detailed in [FPT10].

Workflow patterns support. Currently, most workflow languages support the basic construct of sequence, iterations, splits and joins. However, the interpretation of even these basic constructs is not uniform and it is often unclear how more complex workflow patterns could be supported. For instance, both BPEL and HOCL support a wide variety of complex workflow structures such as sequence, discriminator, synchronization-merge, simple-merge, as detailed in Section 3.2.5. These patterns are applied using some primitives in BPEL, and by the use of some specific chemical rules in HOCL. However, Taverna, because of its data-driven behavior only supports few workflow patterns such as sequence, conditional and parallel split, as the order of execution is specified by data-dependencies with no particular primitives.

To sum up, we consider that any workflow definition can be translated into a chemical program thanks to the data-control driven coordination what we can give to our chemical programs based on the molecular composition analogy. HOCL integrates both the simplicity of data-driven for simple patterns, while supporting complex workflow patterns by the definition of rules for control-driven dependencies.

A.2 Chemical Engine – Generic Rules

In the following, we detail the syntax of the generic rules previously explained in Section 3.2.4 and Section 3.2.5 of Chapter 3. These rules are independent of any workflow definition and enable a decentralized and centralized workflow execution.

Generic Rules

```
import fr.inria.hocl.core.hocli.jms.*;

//--------------------------- HOCL ENGINE -----------------------


//    Centralized execution

let passInfoCentralized = replace chwsSource::String:<"PASS":chwsDest::String:className::String:< ?w >, ?m >, chwsDest::String:<?l>
                by  chwsSource:<m>, chwsDest:<w,l>

in

//    Decentralized execution

// This rule sends the content of PASS molecule to the destination ChWS.

let passInfo = replace "PASS":idChWS::String:< ?w >
                by  "Success_Pass":idChWS:TransferMolecule.put(idChWS,w)

in



// This rule sends the content of PASS molecule to the destination ChWS whether
// one condition is satisfied. --DEPRECATED --

let passInfoCond = replace "PASS":idChWS::String:< ?w > , "COND_PASS":value::int
                by  "Success_Pass_Cond":idChWS:TransferMolecule.put(idChWS,w), "COND_PASS":value
               if (value==1)
in

// This rule sends ones the content of PASS molecule to the destination ChWS
// whether one condition is satisfied, Exclusive-choice.
```

```
let passExcluCond = replace-one "PASS":idChWS::String:< ?w > , "COND_PASS":idChWS::String:value::int
                by  "Success_Pass_Cond":idChWS:TransferMolecule.put(idChWS,w), "COND_PASS":value
               if( (value==1) )
in


// This rule sends the content of PASS molecules to the destination ChWSes whether
//  one condition is satisfied, Multi-choice.

let passInfoMultCond = replace "PASS":idChWS::String:< ?w > , "COND_PASS":idChWS::String:value::int
                by  "Success_Pass_Cond":idChWS:TransferMolecule.put(idChWS,w), "COND_PASS":idChWS:value
               if( (value==1) )
in

// This rule sends the content of CANCEL molecule to the destination CANCEL_ChWS.

let passInfoCancel = replace-one (passInfo=m), "CANCEL":< ?w > , "CANCEL_ChWS":idChWS::String
                       by "Success_Cancel":TransferMolecule.put(idChWS,w)

in

// This rule stores the CONDITION value and the result of one invocation into
// the PASS molecule.

let preparePassCond = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >, "CONDITION":value::int
                    by "PASS":nameRegService:<"COMPLETED":idChWS:< w >, "CONDITION":value>,"CONDITION":value,"RESULT":idChWS:< w >
                    if (value == 1)
in

// This rule stores the result of one invocation into the PASS molecule.

let preparePass = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >
                      by "PASS":nameRegService:<"COMPLETED":idChWS:< w > >,"RESULT":idChWS:< w >
in

// This rule stores the content of the INFORMATION molecule into the PASS molecule.

let prepareInfoInPass = replace-one "INFORMATION":nameRegService::String:< ?w >
                    by "PASS":nameRegService:< w >,passInfo
in

// This rule stores the DISCRIMINATOR molecule and the result of one invocation
// into the PASS molecule.

let discr_preparePass = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >
                     by "PASS":nameRegService:<"DISCRIMINATOR","COMPLETED":idChWS:< w > >
in

// This rule stores the MERGE molecule and the result of one invocation into the PASS molecule.

let sm_preparePass = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >
                       by "PASS":nameRegService:<"MERGE","COMPLETED":idChWS:< w > >
in

// This rule starts the execution of one ChWS producing a INVOKE molecule when
// the MERGE molecule appears into the subsolution.

let sm_setFlag = replace-one "MERGE"
                       by "FLAG_INVOKE"
in

// This rule starts the execution of one ChWS producing a FLAG_INVOKE molecule when
// the DISCRIMINATOR molecule appears into the subsolution.

let discr_setFlag = replace-one "DISCRIMINATOR"
                              by "FLAG_INVOKE"

in

// This rule starts the execution of one ChWS when the COMPLETED molecule appears
// into the subsolution.

let setFlag = replace-one "COMPLETED":idChWS::String:< ?w >
                       by "FLAG_INVOKE","COMPLETED":idChWS:< w >
in

// These rules are used to apply the  Synchronization merge pattern.

let syncMg_preparePass = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >, "WAITFOR":number::int
                by "PASS":nameRegService:<"WAITFOR":number, "COMPLETED":idChWS:< w > >, "WAITFOR":number,"RESULT":idChWS:< w >
in

// These rules wait until all COMPLETED molecules are been received, the number
// of molecules depends of the molecule WAITFOR. These rules are used to apply
// the synchrnoziation merge pattern.

let syncMerge = replace "WAITFOR":number::int, "COMPLETED":idChWS1::String:< ?p >, "SYNCMG_INBOX":< ?w >
                by    "SYNCMG_INBOX":<"COMPLETED":idChWS1:< p >, w>, "WAITFOR":(number-1)
                if  number > 0
in
```

```
let syncMg_setFlag = replace-one "WAITFOR":0
                       by   "FLAG_INVOKE"
in

// These rules are used to apply the Synchronization pattern.

let synchronize = replace "IN":x::int, "COMPLETED":idChWS1::String:< ?p >, "START_INVOCATION":<?w >
                  by "START_INVOCATION":< "COMPLETED":idChWS1:< p >,w >, "IN":(x-1)
                  if x > 0
in
let sync_setFlag = replace-one "IN":0
                                      by "FLAG_INVOKE"
in

// -------------- LOOP ----------

// This rule sends the information to the destination node when the loop condition is true.

let putPassInfo = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >, "CONDITION_LOOP":value::int
                  by "PASS":nameRegService:<"COMPLETED":idChWS:< w > >
                  if (value == 1)
in
// This rule sends the information to the first node involved in the loop when the loop condition is false.

let putPassLoop = replace "DEST_LOOP":nameRegService::String, "CONDITION_LOOP":value::int, "INFORMATION_LOOP":< ?q >
                  by "PASS":nameRegService:< q >, "CONDITION_LOOP":value, "DEST_LOOP":nameRegService
                  if (value == 0)
in

// This rule puts the information into the molecule PASS and it keeps the DEST molecules for the next transfers during the loop.

let loop_preparePass = replace "DEST":nameRegService::String, "RESULT":idChWS::String:< ?w >
                         by "PASS":nameRegService:<"COMPLETED":idChWS:< w > >,"DEST":nameRegService
in


// ----- MULTI-MERGE Workflow Pattern -----
//These rules are used to apply the Multi merge workflow pattern.

let mm_reset = replace-one "RESET":< ?w >, "LOCKED":0, "FLAG_INVOKE"
            by w,"RESET":< w >, "LOCKED":1, "FLAG_INVOKE"
in

let mm_lockReset = replace "LOCKED":1, "RESULT":idChWS::String:< ?w >, "Success_Pass":destination::String:pass::boolean
                by mm_reset, "LOCKED":0
in

let mm_lockReset_End = replace "LOCKED":1, "RESULT":idChWS::String:< ?w >
                     by mm_reset, "LOCKED":0
in


// Two new rules for mm_reset when using a condition.


let mm_lockReset_Cond = replace "LOCKED":0, "RESULT":idChWS::String:< ?w >,
                          "Success_Pass_Cond":destination::String:pass::boolean
                    by mm_reset, "LOCKED":0
in

let mm_lockReset_Condition_End = replace "LOCKED":1, "RESULT":idChWS::String:< ?w >, "CONDITION":1
                        by mm_reset, "LOCKED":0
in



//-------------------------------------------------------------------------------------------------------------
```

A.2.1 BlastReport Workflow Definition

BlastReport is a home-built bioinformatics workflow which retrieves a blast report of a protein in a database given its ID. A chemical workflow definition of this BlastReport workflow expresses all the control and data dependencies, is illustrated on Table A.2.

```
<

"ChWS_1":<
  invoke_fetchBatch,"CALL":"ChWS_1":"http://www.ebi.ac.uk/ws/services/WSDbfetch?wsdl",passInfo,
    "INVOKE":1,"PARAM":<"db":"emblcds","style":"default","ids":"EDL10223.1","format":"fasta">, "DEST":"ChWS_2":"MultisetBlastR",preparePass
 >
,
"ChWS_2":<
  invoke_run,"CALL":"chWS_2":"http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl",passInfo,
  "OPERATION":"run","PARAM":<"program":"blastp","stype":"protein","title":"Proof">,
  (replace-one "COMPLETED":idChWS::String:<"fetchBatchReturn":result::String>, "PARAM":<?w>  by "PARAM":<w,"sequence":result>, "INVOKE":1),
  "DEST":"ChWS_4":"MultisetBlastR","DEST":"ChWS_3":"MultisetBlastR","DEST":"ChWS_5":"MultisetBlastR",preparePass
  >,
"ChWS_3":<
  invoke_getStatus,"CALL":"chWS_3":"http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl",passInfo,
  "OPERATION":"getStatus","PARAM":<>, "DEST":"ChWS_4":"MultisetBlastR",
  (replace-one "COMPLETED":idChWS::String:<"jobId":result::String>, "PARAM":<?w>   by "jobid":result, "INVOKE":1,"PARAM":<w,"jobid":result>),
  (replace "RESULT":"ChWS_3":<"status":result::String>, "jobid":value::String
      by preparePass, "RESULT":"ChWS_3":<"status":result>
      if (result.contains("FINISHED"))),
  (replace "jobid":result::String, "RESULT":"ChWS_3":<"status":result2::String>
      by "INVOKE":1,"jobid":result,"OPERATION":"getStatus", "PARAM":<"jobid":result> if (!result2.contains("FINISHED")))
 >,
"ChWS_4":<
   invoke_getResultTypes,"CALL":"chWS_4":"http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl",passInfo,
   "OPERATION":"getResultTypes","PARAM":<>, ,"DEST":"ChWS_5":"MultisetBlastR",preparePass,
  (replace-one "COMPLETED":idChWS2::String:<"status":"FINISHED">,"COMPLETED":idChWS::String:<"jobId":result::String>, "PARAM":<?w> by "PARAM":<w,"jobId":result>, "INVOKE":1)
 >,
"ChWS_5":<
    invoke_getResult,"CALL":"chWS_5":"http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl",passInfo,
    "OPERATION":"getResult","PARAM":<>,
    (replace-one "COMPLETED":idChWS::String:<"jobId":result::String>, "PARAM":<?w> by "PARAM":<w,"jobid":result> ),
    (replace-one "COMPLETED":idChWS::String:<"ARRAY":<<"identifier":"sequence",?w>,?p>>, "PARAM":<?l> by "PARAM":<l,"type":"sequence">, "INVOKE":1)
 >
>
```

Table A.2: BlastReport chemical workflow definition

A.2.2 Montage Workflow Definition

Montage[1] [32] is a classic astronomical image mosaic workflow processing large images of the sky. The chemical workflow definition modelling the 27 ChWSes and its control and dependencies is illustrated on Tables A.3, A.4 and A.5.

A.2.3 Cardiac Analysis Workflow Definition

CardiacAnalysis is a cardiovascular image analysis workflow which extracts the heart's anatomy from a series of image sequences by applying image processing algorithms, developed by the CREATIS-LRMN biomedical laboratory[2]. The chemical workflow definition modelling the six ChWSes and its control and dependencies is illustrated on Table A.6.

A.3 Workflow Scheduling using HOCL

In this section, we show a set of *generic rules* used for scheduling workflows in our chemically coordinated scheduling system, as detailed in Chapter 5.

A.3.1 Workflow Decomposition

As briefly introduced in Section 5.2.6.2 of Chapter 5, we here present a detailed version of the *generic rules* needed for a workflow decomposition. Considering any chemical workflow specification, the following rules decompose it in levels, producing new molecules that represent these levels.

To do that decomposition in ascending order, we need some additional rules which are given in Algorithm 16. Firstly, the chemical engine launchs the *exitNodes* and *noExitNodes* rules, responsible for the identification of the exit and leaf nodes in a workflow. Once, we discovered these nodes, the *initGetLevels* and *getLevels* rules iterate over all the nodes until the source node associating them to a specific level. Note that, the *getNumLevels* rule transforms the descendending into ascending order for the id's (denoted as *num* Line 19.11) of each level. Furthermore, some reaction rules are included into the main solution in order to execute in sequence some of the rules previously defined.

[1] http://montage.ipac.caltech.edu/
[2] http://www.creatis.insa-lyon.fr/site/

```
<
"ChWS_1":<
  "DEST":"ChWS_7","DEST":"ChWS_9","DEST":"ChWS_12", passInfo,preparePass,"FLAG_INVOKE","OPERATION":"mProjectPP","CALL":"ChWS_1":"http://localhost:8080/ode/processes/MontageService?wsdl",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1440186.fits", "param2":"p2mass-atlas-990502s-j1440186.fits" , "param3":"region_20080505_143233_14944.hdr">
 >,
"ChWS_2":<
  "DEST":"ChWS_7","DEST":"ChWS_8", passInfo,preparePass, "FLAG_INVOKE","OPERATION":"mProjectPP","CALL":"ChWS_2":"http://localhost:8080/ode/processes/MontageService?wsdl",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1440198.fits", "param2":"p2mass-atlas-990502s-j1440198.fits" , "param3":"region_20080505_143233_14944.hdr">
  >,
"ChWS_3":<
  "DEST":"ChWS_9","DEST":"ChWS_11","DEST":"ChWS_10", passInfo,preparePass, "FLAG_INVOKE","OPERATION":"mProjectPP","CALL":"ChWS_3":"http://localhost:8080/ode/processes/MontageService?wsdl",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1430092.fits", "param2":"p2mass-atlas-990502s-j1430092.fits" , "param3":"region_20080505_143233_14944.hdr">
 >,
"ChWS_4":<
  "DEST":"ChWS_15","DEST":"ChWS_13", passInfo,preparePass,"FLAG_INVOKE","OPERATION":"mProjectPP","CALL":"ChWS_4":"http://localhost:8080/ode/processes/MontageService?wsdl",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1420198.fits", "param2":"p2mass-atlas-990502s-j1420198.fits" , "param3":"region_20080505_143233_14944.hdr">
 >,
"ChWS_5":<
  "DEST":"ChWS_14","DEST":"ChWS_13","DEST":"ChWS_10", passInfo,preparePass,"FLAG_INVOKE","OPERATION":"mProjectPP","CALL":"ChWS_5":"http://localhost:8080/ode/processes/MontageService?wsdl",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1420186.fits", "param2":"p2mass-atlas-990502s-j1420186.fits" , "param3":"region_20080505_143233_14944.hdr">
 >,
"ChWS_6":<
  "DEST":"ChWS_8","DEST":"ChWS_11","DEST":"ChWS_12","DEST":"ChWS_14","DEST":"ChWS_15", passInfo,preparePass,"FLAG_INVOKE","OPERATION":"mProjectPP",
  "PARAM":<"param0":"1.00638", "param1":"/2mass-atlas-990502s-j1430080.fits", "param2":"p2mass-atlas-990502s-j1430080.fits" , "param3":"region_20080505_143233_14944.hdr">,
  "CALL":"ChWS_6":"http://localhost:8080/ode/processes/MontageService?wsdl",
 >,
"ChWS_7":<
   "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_7":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
      by "PARAM":<"param0":"fit.000001.000003.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000001.000003.fits" ,"param4":"region_20080505_143233_14944.hdr">)
 >,
"ChWS_8":<
  "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_8":"http://localhost:8080/ode/processes/MontageService?wsdl",
  (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
    by "PARAM":<"param0":"fit.000001.000006.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000001.000006.fits" ,"param4":"region_20080505_143233_14944.hdr">)
  >,
"ChWS_9":<
  "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_9":"http://localhost:8080/ode/processes/MontageService?wsdl",
  (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
    by "PARAM":<"param0":"fit.000002.000003.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000002.000003.fits" ,"param4":"region_20080505_143233_14944.hdr">)
 >,
"ChWS_10":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_10":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
      by "PARAM":<"param0":"fit.000002.000004.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000002.000004.fits" ,"param4":"region_20080505_143233_14944.hdr">)
  >,
"ChWS_11":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_11":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
      by "PARAM":<"param0":"fit.000002.000006.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000002.000006.fits" ,"param4":"region_20080505_143233_14944.hdr">)
   >,
```

Table A.3: Montage chemical workflow definition – part.1.

```
"ChWS_12":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_12":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
      by "PARAM":<"param0":"fit.000003.000006.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000003.000006.fits" ,"param4":"region_20080505_143233_14944.hdr">)
  >,

"ChWS_13":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, "DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_13":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
  by "PARAM":<"param0":"fit.000004.000005.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000004.000005.fits" ,"param4":"region_20080505_143233_14944.hdr">)
  >,

"ChWS_14":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2, ,"DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_14":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
by "PARAM":<"param0":"fit.000004.000006.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000004.000006.fits" ,"param4":"region_20080505_143233_14944.hdr">)
   >,
"ChWS_15":<
    "OPERATION":"mDiffFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":2,"DEST":"ChWS_16",passInfo,preparePass,"CALL":"ChWS_15":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "START_INVOCATION":<"COMPLETED":idChWS::String:<"return":fileName1::String>,"COMPLETED":idChWS::String:<"return":fileName2::String>>
by "PARAM":<"param0":"fit.000005.000006.txt", "param1":fileName1, "param2":fileName2, "param3":"diff.000005.000006.fits" ,"param4":"region_20080505_143233_14944.hdr">)
  >,
"ChWS_16":<
    "OPERATION":"mConcatFit",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":9, ,passInfo,preparePass,"CALL":"ChWS_16":"http://localhost:8080/ode/processes/MontageService?wsdl",
    "DEST":"ChWS_17","PARAM":<"param0":"statfile_20080505_143233_14944.tbl", "param1":"fits.tbl","param2":"/home/hectorj2f/temp/exampleProof">
  >,
"ChWS_17":<
    "OPERATION":"mBgModel","FLAG_INVOKE", "DEST":"ChWS_22","DEST":"ChWS_23",passInfo,preparePass,"CALL":"ChWS_17":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String> by "PARAM":<"param0":"100000", "param1":"pimages_20080505_143233_14944.tbl", "param2":fileName, "param3":"corrections.tbl">),
     "DEST":"ChWS_18","DEST":"ChWS_19","DEST":"ChWS_20","DEST":"ChWS_21"
    >,
"ChWS_18":<
    "OPERATION":"mBackground","FLAG_INVOKE","DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_18":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1440198.fits", "param2":"c2mass-atlas-990502s-j1440198.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
   >,
"ChWS_19":<
    "OPERATION":"mBackground","FLAG_INVOKE", "DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_19":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1430092.fits", "param2":"c2mass-atlas-990502s-j1430092.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
  >,
"ChWS_20":<
    "OPERATION":"mBackground","FLAG_INVOKE", ,"DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_20":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1440186.fits", "param2":"c2mass-atlas-990502s-j1440186.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
  >,
"ChWS_21":<
    "OPERATION":"mBackground","FLAG_INVOKE", ,"DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_21":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1420186.fits", "param2":"c2mass-atlas-990502s-j1420186.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
  >,
```

Table A.4: Montage chemical workflow definition – part.2.

```
"ChWS_22":<
    "OPERATION":"mBackground","FLAG_INVOKE", "DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_22":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1420198.fits", "param2":"c2mass-atlas-990502s-j1420198.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
  >,

"ChWS_23":<
    "OPERATION":"mBackground","FLAG_INVOKE", ,"DEST":"ChWS_24",passInfo,preparePass,"CALL":"ChWS_23":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "PARAM":<"param0":"", "param1":"p2mass-atlas-990502s-j1430080.fits", "param2":"c2mass-atlas-990502s-j1430080.fits", "param3":"pimages_20080505_143233_14944.tbl", "param4":fileName>)
  >,

"ChWS_24":<
    "OPERATION":"mImgtbl",synchronize, sync_setFlag, "START_INVOCATION":<>,"IN":6, "DEST":"ChWS_25",passInfo,preparePass,
    "PARAM":<"param0":"", "param1":"cimages_20080505_143233_14944.tbl","param2":"newimages.tbl">,"CALL":"ChWS_24":"http://localhost:8080/ode/processes/MontageService?wsdl"
  >,
"ChWS_25":<
    "OPERATION":"mAdd", "DEST":"ChWS_26",passInfo,preparePass,"CALL":"ChWS_25":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "FLAG_INVOKE","PARAM":<"param0":"","param1":fileName, "param2":"region_20080505_143233_14944.hdr","param3":"mosaic_20080505_143233_14944.fits">)
  >,
"ChWS_26":<
    "OPERATION":"mShrink", "DEST":"ChWS_27",passInfo,preparePass,"CALL":"ChWS_26":"http://localhost:8080/ode/processes/MontageService?wsdl",
    (replace "COMPLETED":idChWS::String:<"return":fileName::String> by "FLAG_INVOKE","PARAM":<"param0":fileName, "param1":"shrunken_20080505_143233_14944.fits">)
  >,
"ChWS_27":<
    "OPERATION":"mJpeg","CALL":"ChWS_27":"http://localhost:8080/ode/processes/MontageService?wsdl"
    (replace "COMPLETED":idChWS::String:<"return":fileName::String>
      by "FLAG_INVOKE","PARAM":<"param0":1, "param1":"-gray", "param2":fileName, "param3":"-1.5s", "param4":"60s", "param5":"montage.jpg">),
  >
>
```

Table A.5: Montage chemical workflow definition – part.3.

```
<
"ChWS_1":<
  "DEST":"ChWS_2",passInfo,preparePass,"CALL":"ChWS_1":"http://adonis-5:8080/ode/processes/DICOMAnalyzerService?wsdl",
  "PARAM":<"param0":"/root/MpiReg/bin/", "param1":"/root/MpiReg/dataold", "param2":"/root/MpiReg/dataold/sorted">,"FLAG_INVOKE"

 >,
"ChWS_2":<
  (  replace "COMPLETED":idChWS::String:<"ARRAY":<"return":dir::String,?w>>
      by "COMPLETED":idChWS:<"ARRAY":<w>>,"FLAG_INVOKE",
  "PARAM":<"param0":"/root/MpiReg/bin/", "param1":dir, "param2":20 , "param3":71 ,"param4":142, "param5":169>
  ),
  "DEST":"ChWS_3",passInfo,loop_preparePass,"CALL":"ChWS_2":"http://adonis-5:8080/ode/processes/ImageCropService?wsdl"
>,
"ChWS_3":<
   (
      replace "COMPLETED":idChWS::String:<"ARRAY":<"return":dir::String,?w>>
    by "COMPLETED":idChWS:<"ARRAY":<w>>,"FLAG_INVOKE","PARAM":<"param0":"/root/MpiReg/bin/", "param1":dir,"param2":"Heart.mhd">
   ),
   "CALL":"ChWS_3":"http://adonis-5:8080/ode/processes/InterpolationService?wsdl", "DEST":"ChWS_4", passInfo, loop_preparePass
>,
"ChWS_4":<
   ( replace "COMPLETED":idChWS::String:<"return":pathImage::String>
      by "DEST":"ChWS_5","DEST":"ChWS_6","FLAG_INVOKE","PARAM":<"param0":"/root/MpiReg/bin/",
      "param1":(pathImage.substring(0,pathImage.lastIndexOf("/"))),
      "param2":(pathImage.substring(pathImage.lastIndexOf("/")+1,pathImage.length())), "param3":"/root/MpiReg/config.init">
   ),
   "CALL":"ChWS_4":"http://adonis-5:8080/ode/processes/ImagePyramidDecomService?wsdl",  passInfo, loop_preparePass
 >,
"ChWS_5":<
   ( replace "COMPLETED":idChWS::String:<"ARRAY":<"return":pathFile0::String,?w>>
      by "COMPLETED":idChWS:<"ARRAY":<w>>, "FLAG_INVOKE","PARAM":<"param0":"/root/MpiReg/bin/", "param1":pathFile0>
   ),
    "CALL":"ChWS_5":"http://adonis-5:8080/ode/processes/GradientComputingService?wsdl"
 >,
"ChWS_6":<
    ( replace "COMPLETED":idChWS::String:<"ARRAY":<"return":pathFile0::String,?w>>
      by "COMPLETED":idChWS:<"ARRAY":<w>>,"FLAG_INVOKE","PARAM":<"param0":"/root/MpiReg/bin/","param1":pathFile0,"param2":0,"param3":4,"param4":"0.1">
    )
  , "CALL":"ChWS_6":"http://adonis-5:8080/ode/processes/BorderDetectionService?wsdl"
 >
>
```

Table A.6: CardiacAnalysis chemical workflow definition.

Algorithm 16 Generic rules — Workflow Decomposition.

```
19.01 let exitNodes = replace idWS:⟨ ?w ⟩, It_levels:num, Nodes:⟨ ?x ⟩
19.02                 by Levels:⟨ Level:num:⟨ idWS ⟩, It_Levels:(num-1), Level:(num-1):⟨ ⟩, Nodes:⟨ x ⟩
19.03 let noExitNodes = replace idWS:⟨ ?p, Dest:destination ⟩, Nodes:⟨ ?w ⟩
19.04                 by Nodes:⟨ idWS:⟨ p, Dest:destination ⟩, w ⟩,
19.05 let getLevels = replace It_Levels:num, Exit_Node:⟨ dest, ?m ⟩, Level:num:⟨ ?x ⟩,
19.06                       Nodes:⟨ idWS:⟨ ?p,Dest:destination ,?ω ⟩, ?h ⟩
19.07                 by Level:num:⟨ idWS, x ⟩, Exit_Node:∠ dest, m ⟩, It_Levels:num, Nodes:⟨ h ⟩
19.08                 if (dest==destination)
19.09 let initGetLevels = replace ⟨ Nodes:⟨ idWS:⟨ ?p ⟩, ?x ⟩, ?ω ⟩
19.10                 by ⟨ Nodes:⟨ idWS:⟨ p ⟩, x ⟩, getLevels, ω ⟩
19.11 let getNumLevels = replace Rest:x, Levels:⟨ Level:num:⟨ ?k ⟩, ?g ⟩, It_Levels:value

19.12                 by Rest:x, Level:(num-x):⟨ k ⟩, Levels:⟨ g ⟩, It_Levels:(value+1)

                        — MAIN SOLUTION CONTENT —
19.13 ⟨
19.14       ...,
19.15 (replace-one ⟨ noExitNodes=s,?ω ⟩ by ⟨w, exitNodes, getLevels⟩),initGetLevels,
19.16 (replace-one ⟨getLevels=s, Level:num:⟨?l⟩, Exit_Node:⟨?m⟩,It_Levels:num, Levels:⟨?w⟩, ?k⟩
19.17       by ⟨ k, Exit_Node:⟨ l ⟩, Levels:⟨Level:num:⟨ l ⟩, w⟩,
19.18             It_Levels:(num-1), Level:(num-1):⟨ ⟩ ⟩),
19.19 (replace-one initGetLevels=s, ⟨Nodes:⟨"END"⟩, Level:num:⟨?l⟩, ?w⟩ by w ),
19.20 (replace-one Exit_Node:⟨x, ?w⟩, Levels:⟨Level:num:⟨?k⟩, ?g⟩, It_Levels:value
19.21       by Rest:num, Level:0:⟨k⟩, Levels:⟨g, "END"⟩, It_Levels:0),
19.22 getNumLevels,
19.23 (replace-one getNumLevels=s, Rest:x, Levels:⟨"END"⟩, ?l by l)
19.24 ⟩
```

A.3.2 Opportunistic Load Balancing

The OLB heuristic assigns each task to the first available resource without considering any time estimations by running this task on that resource.

Chemical implementation. In Algorithm 17, the *mapOpploadBalancing* rule associates each task to the first available resource in the system.

Algorithm 17 Generic rules — OLB

```
20.01 let mapOpploadBalancing = replace Task:idT:weight, Res:idR:⟨cpuLoad,memUsed,numProc,?ω ⟩
20.02                   by Res_Task:idT:idR
```

A.3.3 Max-min

The Max-min heuristic finds for each eligible task, the resource that gives the *maximum completion* time for a task.

Chemical implementation. In Algorithm 18, first, the *sortedTaskMax* and *sortedTaskMaxEnd* rules obtain the task with the minimum weight respectively. Then, the *initResList* and *resetResList* rules iterate task by task to associate them to each resource. Next, the *calcEstMaxmin* rule calculates the *maximum completion* time of each ready task on each resource. Finally, the reactions rule into the main solution selects the mapping task-resource that gives the *maximum completion* time.

Algorithm 18 Generic rules — Max-min

```
21.01 let resetResList = replace TASK:idT:cost, SORTED_TASKS:⟨TASK:idTX:costX, ?ω ⟩,
21.02                            RES_LIST:⟨ RES:idR:⟨?m⟩, ?n⟩,RES_MAPP_LIST:⟨"END"⟩,
21.03                            RES_TASK:idT:⟨idR:costRes, "END"⟩
21.04                 by RES_TASK:idT:⟨idR:costRes⟩,RES_MAPP_LIST:⟨n, "END"⟩, RES_LIST:⟨n⟩,
21.05                            TASK:idTX:costX, RES_TASK:idTX:⟨"END"⟩, SORTED_TASKS:⟨w⟩
21.06 let initResList = replace-one SORTED_TASKS:⟨TASK:idT:cost,?ω ⟩, RES_LIST:⟨?p⟩
21.07                 by SORTED_TASKS:⟨w⟩, TASK:idT:cost, resetResList, RES_TASK:idT:⟨"END"⟩,
21.08                            RES_MAPP_LIST:⟨p, "END"⟩, RES_LIST:⟨p⟩
21.09 let calcEstMaxmin = replace TASK:idT:cost, RES_TASK:idT:⟨ ?o ⟩,
21.10                            RES_MAPP_LIST:⟨ RES:idR:⟨cpuLoad,memUsed,numProc,?w⟩, ?l⟩
21.11                 by TASK:idT:cost, RES_MAPP_LIST:⟨l⟩,
21.12                       RES_TASK:idT:⟨ idR:calculateETC_Maxmin(idT,idR,cpu,mem,nP, cost), o⟩
21.13 let mapMaxmin = replace RES_TASK:idT:⟨idRX:costX, idRY:costY, ?l⟩
21.14                 by RES_TASK:idT:⟨idRX:costX, l⟩
21.15                 if (costX >= costY)
21.16 let sortedTaskMax = replace TASKS:⟨TASK:idX:costX, TASK:idY:costY, ?ω ⟩, SORTED_TASKS:⟨ ?p ⟩
21.17                 by TASKS:⟨ TASK:idY:costY, ω ⟩, SORTED_TASKS:⟨ TASK:idX:costX, p⟩
21.18                 if (costX >= costY)
21.19 let sortedTaskMaxEnd = replace TASKS:⟨TASK:idT:costX⟩, SORTED_TASKS::⟨ ?p ⟩
21.20                 by SORTED_TASKS:⟨p, TASK:idX:costX, "END“⟩, initResList

                        — MAIN SOLUTION CONTENT —
21.21 ⟨
21.22         ...,
21.23         mapMaxmin, calcEstMaxmin, sortedTaskMax, sortedTaskMaxEnd,
21.24         (replace-one TASK:idT:cost, SORTED_TASKS:⟨"END"⟩, RES_MAPP_LIST:⟨"END"⟩,
21.25               RES_TASK:idT:⟨idR:costRes, "END"⟩ by  RES_TASK:idT:⟨ idR:costRes ⟩ )
21.26 ⟩
```

A.3.4 Min-min

The Min-min heuristic finds for each eligible task (ready to be executed), the resource that gives the *minimum completion* time for this task.

Chemical implementation. In Algorithm 19, first, the *sortedTaskMin* and *sortedTaskMinEnd* rules obtain the task with the minimum weight respectively. Then, the *initResList* and *resetResList* rules iterate task by task to associate them to each resouce. Next, the *calcEstMinmin* rule calculates the *minimum completion* time of each ready task on each resource. Finally, the reaction rule into the main solution selects the mapping task-resource that gives the *minimum completion* time.

Algorithm 19 Generic rules — Min-min

```
22.01 let resetResList = replace Task:idT:cost, Sorted_Tasks:⟨Task:idTX:costX, ?ω ⟩,
22.02                            Res_List:⟨ Res:idR:⟨?m⟩, ?n⟩,Res_Mapp_List:⟨"END"⟩,
22.03                            Res_Task:idT:⟨idR:costRes, "END"⟩
22.04             by Res_Task:idT:⟨idR:costRes⟩,Res_Mapp_List:⟨n, "END"⟩, Res_List:⟨n⟩,
22.05                            Task:idTX:costX, Res_Task:idTX:⟨"END"⟩, Sorted_Tasks:⟨w⟩
22.06 let initResList = replace-one Sorted_Tasks:⟨Task:idT:cost,?ω ⟩, Res_List:⟨?p⟩
22.07             by Sorted_Tasks:⟨w⟩, Task:idT:cost, resetResList, Res_Task:idT:⟨"END"⟩,
22.08                            Res_Mapp_List:⟨p, "END"⟩, Res_List:⟨p⟩
22.09 let calcEstMinmin = replace Task:idT:cost, Res_Task:idT:⟨ ?o ⟩,
22.10                            Res_Mapp_List:⟨ Res:idR:⟨cpuLoad,memUsed,numProc,?w⟩, ?l⟩
22.11             by Task:idT:cost, Res_Mapp_List:⟨l⟩,
22.12                    Res_Task:idT:⟨ idR:calculateETC_Minmin(idT,idR,cpu,mem,nP, cost), o⟩
22.13 let mapMinmin = replace Res_Task:idT:⟨idRX:costX, idRY:costY, ?l⟩
22.14             by Res_Task:idT:⟨idRX:costX, l⟩
22.15             if (costX <= costY)
22.16 let sortedTaskMin = replace Tasks:⟨Task:idX:costX, Task:idY:costY, ?ω ⟩, Sorted_Tasks:⟨ ?p ⟩
22.17             by Tasks:⟨ Task:idX:costX, ω ⟩, Sorted_Tasks:⟨ Task:idY:costY, p⟩
22.18             if (costX <= costY)
22.19 let sortedTaskMinEnd = replace Tasks:⟨Task:idT:costX⟩, Sorted_Tasks::⟨ ?p ⟩
22.20             by Sorted_Tasks:⟨p, Task:idX:costX, "END“⟩, initResList

                        — MAIN SOLUTION CONTENT —
22.21 ⟨
22.22       ...,
22.23       mapMinmin, calcEstMinmin, sortedTaskMin, sortedTaskMinEnd,
22.24       (replace-one Task:idT:cost, Sorted_Tasks:⟨"END"⟩, Res_Mapp_List:⟨"END"⟩,
22.25             Res_Task:idT:⟨idR:costRes, "END"⟩ by  Res_Task:idT:⟨ idR:costRes ⟩ )
22.26 ⟩
```

A.3.5 Levelized Minimun Time

This heuristic consists of two phases: level sorting and *Min-Time*. First, the LMT algorithm clusters the tasks that have to be executed in parallel, and then order these tasks based on their data and control dependencies, *i.e.*, level by level. Second, each task is matched to the best available resource, in other words, to the resource that gives the *minimum completion* time for this task. To do that, this algorithm calculates the average execution time (processing and communication cost) of each task across all the available resources. Finally, this algorithm assigns each task to the best available resource, in other

words, each task is matched to the resource on which it executes fastest, as suggested on Algorithm 20.

Chemical implementation. First, the *getLevels* rule calculates the communication cost to reach the nodes, and produces a molecule that contains all the levels with their tasks, as illustrated in Algorithm 20. Second, the *getTaskLevel*, *initResList* and *incrementLevel* rules iterate the lists of levels, extracting one by one the tasks from each level and associating each task to the list of available resources. Third, the *scheddTask* rule produces a molecule, with the form RES_TASK:*idT:*⟨ *idR1:value1, idR2:value2, ...*⟩ containing a list that represents the different completion time of a specific task *idT* for each one of the available resources (from the previous list). Finally, the *mapLmt* rule iterates such list of completion times consuming all the molecules except that one with the minimun time. This resulting molecule represents the resource on which the task *idT* will be executed.

Algorithm 20 Generic rules — LMT

```
23.01 let getLevels = replace-one EDGES:⟨ ?w⟩
23.02                 by LEVELS:LevelWorkflowAnalysis.getLevelTasks("EDGE",w)
23.03 let getTaskLevel = replace-one IT_LEVELS:num, LEVELS:⟨ LEVEL:num:⟨ task, ?m, "END"⟩, ?ω ⟩,
23.04                                RES_LIST:⟨ ?l ⟩
23.05                 by IT_LEVELS:num, RES_LIST:⟨ l ⟩, LEVELS:⟨ LEVEL:num:⟨ m, "END" ⟩, w⟩,
23.06                                RES_MAPP_LIST:⟨l,"END"⟩, RES_TASK:task:⟨"END"⟩
23.07 let incrementLevel = replace-one IT_LEVELS:num, LEVELS:⟨ LEVEL:num:⟨"END"⟩, ?w⟩
23.08                 by LEVELS:⟨ ω ⟩, IT_LEVELS:(num+1)
23.09 let initResList = replace TASK:idT:cost,RES_MAPP_LIST:⟨ "END" ⟩,
23.10                                RES_LIST:⟨ RES:idR:⟨ ?ω ⟩, ?p ⟩, RES_TASK:idT:⟨ idR:costRes, "END"⟩
23.11                 by RES_TASK:idT:⟨ idR:costRes ⟩, incrementLevel, RES_LIST:⟨ p ⟩, getTaskLevel
23.12 let scheddTask = replace TASK:idT:cost, RES_TASK:idT:⟨ ?o ⟩,
23.13                                RES_MAPP_LIST:⟨ RES:idR⟨ cpuLoad,memUsed,numProc,?ω ⟩, ?p ⟩
23.14                 by TASK:idT:cost, RES_MAPP_LIST:⟨ p ⟩
23.15                                RES_TASK:idT:⟨ idR:calcETC_Minmin(idT,idR,cpu,mem,nP, cost), o⟩,
23.16 let mapLmt = replace RES_TASK:idT:⟨ idRX:costX, idRY:costY, ?l ⟩
23.17                 by RES_TASK:idT:⟨ idRX:costX, l⟩
23.18                         if (costX <= costY)

                        — MAIN SOLUTION CONTENT —
23.19 ⟨
23.20         ...,
23.21         getLevels,getTaskLevel,initResList,scheddTask, mapLmt
23.22 ⟩
```

A.3.6 Heterogeneous Earliest-Finish-Time

This is a three phase algorithm. First, a priority value is associated to each task based on the computation and communication costs to reach the exit node from the position (current level) of each task. Second, the resulting list of tasks is sorted by non-increasing

order of their values. Occasionally, if two tasks have the same priority values, one of them is selected randomly. Third, the algorithm assigns each eligible task to the available resource that gives the *minimum computation* time for scheduling. This heuristic is one of most used among the workflow scheduling systems.

Chemical implementation. First, the *communicationCost* rule calculates the communication cost to reach the nodes (denoted as Tasks in Algorithm 21). Second, the *calcRank* and *getFinalRank* produces a molecule representing a list of rank values for each task. Next, the *calcMaxRank*, *getMaxRank* and *initMaxRank* rules iterate on the list of ranks values, denoted by the molecule with the form RANKS:⟨ *?m* ⟩ where *?m* is the list of rank values, and extract in each iteration the task molecule with the maximum rank value. Finally, the *scheddTaskIni*, *mapHelft* and *initResList* rules produce a molecule that represents the mapping of a task to the available resource that gives the *minimum computation* time. This new molecule has the form RES_TASK:*idT*:⟨*idR*:*compCostRes*⟩ where *idT* and *idR* are the identifiers for the task and resource, respectively.

Algorithm 21 Generic rules — HEFT

```
24.01 let communicationCost = replace EDGE:origen:dest:weight, RATE:value, COM_COST:⟨ ?w ⟩
24.02                 by COM_COST:⟨ (origen:dest:(weight/value) ), ω ⟩, RATE:value
24.03 let calcRank = replace COM_COST:⟨ origen:dest:cost,?ω ⟩, RANK:dest:value, TASK:origen:compCost
24.04                 by COM_COST:⟨ ω ⟩, RANK:origen:(compCost+(cost+value)), TASK:origen:compCost,
24.05                       RANK:dest:value
24.06 let getFinalRank = replace RANK:origen:costA, RANK:origen:costB
24.07                 by RANK:origen:costA
24.08                 if (costA <= costB)
24.09 let calcMaxRank = replace RANK:a:costA, RANK:b:costB
24.10                 by RANK:b:costB
24.11                 if (costA <= costB)
24.12 let getMaxRank = replace-one ⟨calcMaxRank=s,RANK:a:costA, ?l ⟩,
24.13                       RANKS:⟨ RANK:a:costA, TASK:a:compCost, ?m ⟩
24.14                 by RANKS:⟨ m ⟩, SCHEDULER:⟨TASK:a:compCost, RANK:a:costA⟩
24.15 let initMaxRank = replace-one RANKS:⟨ ?m ⟩
24.16                 by ⟨ m, calcMaxRank⟩, RANKS:⟨ m ⟩ , getMaxRank
24.17 let scheddTask = replace TASK:a:compCost, RES_TASK:a:⟨ ?o ⟩
24.18                       RES_MAPP_LIST:⟨RES:idR:⟨cpuLoad,memUsed,numProc,?w⟩, ?p ⟩
24.19       by TASK:a:compCost, RES_TASK:a:⟨idR:calcETC_Minmin(a,idR,cpu,mem,nP, compCost), o⟩,
24.20                       RES_MAPP_LIST:⟨p⟩
24.21 let scheddTaskIni = replace-one SCHEDULER:⟨TASK:idT:compCost,RANK:idT:costA⟩, RES_LIST:⟨?w⟩
24.22                 by TASK:idT:compCost, scheddTask, RES_MAPP_LIST:⟨w,"END"⟩, RES_LIST:⟨w⟩,
24.23                       RES_TASK:idT:⟨"END"⟩
24.24 let initResList = replace RES_MAPP_LIST:⟨"END"⟩, TASK:idT:compCost, scheddTask=s,
24.25                       RES_LIST:⟨RES:idR:⟨?w⟩, ?p⟩, RES_TASK:idT:⟨idR:compCostRes, "END"⟩
24.26                 by scheddTaskIni, initMaxRank, RES_TASK:idT:⟨idR:compCostRes⟩,RES_LIST:⟨p⟩
24.27 let mapHelft = replace RES_TASK:idT:⟨idRX:compCostX, idRY:compCostY, ?l⟩
24.28                 by RES_TASK:idT:⟨idRX:compCostX, l⟩
24.29                 if (compCostX <= compCostY)

                        — MAIN SOLUTION CONTENT —
24.30 ⟨
24.31 ...,
24.32 ( replace-one ⟨ communicationCost=s,calcRank=h,getFinalRank=q, RATE:8, COM_COST:⟨"END"⟩, ?ω ⟩
24.33       by ⟨ ω, calcMaxRank⟩, RANKS:⟨ ω, "END" ⟩ ),
24.34 ( replace-one ⟨ calcMaxRank=s, RANK:a:costA, ?l ⟩, RANKS:⟨ RANK:a:costA, TASK:a:compCost, ?m ⟩
24.35       by RANKS:⟨ m ⟩, SCHEDULER:⟨ TASK:a:compCost, RANK:a:costA ⟩),
24.36 ...,
24.37 ( replace-one RANKS:⟨ "END" ⟩, RES_LIST:⟨ ?l ⟩ by  RES_LIST:⟨ l ⟩),
24.38 scheddTaskIni, mapHelft, initResList
24.39 ⟩
```

Index

A

Abstract workflow 25
Atomic capture 45

B

Black box 17

C

Choreography 19
Coarse-grained 18

E

Encapsulation 18

F

Fine-grained 18
Flexibility 18

G

Generic rules 56

H

Higher-order rules 46
HLPN 23

I

Inertia state 43

K

Kahn networks 51

L

Linda 39
Loose coupling 18

M

Molecular composition 56
Multiset 43

N

N-shot rules 46
Non-deterministic 43

O

One-shot rules 45
Orchestration 19

R

Reaction condition 45

S

Service 15
Stateless 18

T

Task 18
Tight coupling 18
Tuple 47
Tuplespace 39
Type 47

W

Web service 15
Workflow 17
Workflow management system (WMS) 27
Workflow pattern 23